HUMBOLDT-UNIVERSITÄT ZU BERLIN

Kultur-, Sozial- und Bildungswissenschaftliche Fakultät
Institut für Musikwissenschaft und Medienwissenschaft

Masterarbeit zur Erlangung des akademischen Grades
Master of Arts (M.A.) im Fach Medienwissenschaft

„We deserve to have control of our own computing."

Medienwissenschaftliche Überlegungen zur Freiheit von Software und deren Nutzung

„We deserve to have control of our own computing."
Media theoretical reflections on the freedom of software and its utilization

vorgelegt von
Janine Flohr
575954
janine.flohr@gmx.net

1. Gutachter: Dr. Stefan Höltgen
2. Gutachter: Prof. Dr. Wolfgang Ernst

Berlin, März 2018
[überarbeitet im August 2018]

Inhaltsverzeichnis

1 Einleitung

Richard Stallman, auch bekannt als der letzte unter den wahren Hackern,[1] rief vor über 30 Jahren die sogenannte *Free Software Foundation* ins Leben, mit der er es sich zur Aufgabe gemacht hat, die Verbreitung von Freier Software voranzutreiben. Seinem Verständnis nach ist eine Software dann frei, wenn sie in ihrer Ausführung, Verbreitung und Veränderbarkeit unabhängig von ihrem Urheber ist. Als Prämisse hierfür gilt der uneingeschränkte Zugang zum Quelltext eines Programms. Mit dem bereits im Titel genannten Zitat Stallmans: „We deserve to have control of our own computing […]"[2] kann knapp zusammengefasst werden, was die Bewegung motiviert: die Forderung, dass jedem Nutzer[3] die Freiheit und die Kontrolle über die eigenen Computeraktivitäten zustehen. Für Stallman ist die Umsetzung dessen klar:

> […] how can we win this control? By rejecting nonfree software on the computers we own or regularly use […]. By developing free software (for those of us who are programmers). By refusing to develop or promote nonfree software […]. By spreading these ideas to others.[4]

Es liegt damit auf der Hand, dass Freie Software mehr ist als bloß die lizenzrechtliche Form eines Computerprogramms. Ihre Bedeutung reicht von ethischen Werten bis hin zum sozialpolitischen Protest. Denn Freiheit ist stets auch eine Frage der Macht und das Thema in der heutigen Zeit aktueller denn je. „We live in an age transformed by computing."[5], ließ Paul Ceruzzi bereits vor über zehn Jahren verlauten. Eine Abkehr von dem Phänomen Computer ist heute keinesfalls erkennbar; spätestens seit der Verbreitung des Smartphones ist die allumfassende Vernetzung zur Realität geworden.

---

[1] Diese Meinung vertritt Steven Levy in seiner ausführlichen Abhandlung zur Geschichte der Hackerkultur. Wenngleich Stallman nicht wirklich als der letzte aller Hacker zu verstehen ist, so erfüllt er in Levys Augen doch als Letzter die Anforderungen der Hacker Ethik und bringt eine Ära dessen zu Ende, was Levy als das wahre Hackertum ansieht – typisch hierfür sind unter anderem die Ablehnung des Konkurrenzdenkens und ein freier Wissenstransfer. Vgl. hierzu: Levy, Steven: *Hackers. Heroes of the computer revolution*. New York: Anchor Press/Doubleday 1984.

[2] Stallman, Richard: „Free Software Is Even More Important Now", in: https://www.gnu.org/philosophy/free-software-even-more-important.en.html (Abrufdatum: 20.12.2017).
Der Begriff des *Computing* kann nur schwer ins Deutsche übersetzt werden. Eine Definition von Paul Ceruzzi kann hier weiterhelfen: „Computers were invented to ‚compute': to solve ‚complex mathematical problems,' as the dictionary still defines that word." Und ferner: „The word ‚computer' originally meant a *person* who solved equations; it was only around 1945 that the name was carried over to machinery." Ceruzzi, Paul E.: *A history of modern computing*. Cambridge/London: MIT Press 2003, S. 1.

[3] Aus Gründen der Lesbarkeit wird in der vorliegenden Arbeit auf eine genderneutrale Schreibweise verzichtet. Der generische Ausdruck „der Nutzer" soll daher stets auch „die Nutzerin" implizieren. Gleiches gilt für Begriffe wie „der User", „der Anwender", etc.

[4] Stallman: „Free Software Is Even More Important Now".

[5] Ceruzzi: *A history of modern computing*, S. 2.

Maßgeblich für die Nutzung aller Computer sind ihre Programme; erst sie ermöglichen es, die Maschine tatsächlich zu *benutzen*, d.h. sie auf ein bestimmtes Ziel hin zu instruieren. Umgekehrt ist eine Software ohne die Hardware, auf der sie läuft, überflüssig, erfüllt sie doch einzig den Zweck, von einem Computer ausgeführt zu werden. Es erscheint demnach erforderlich, Software stets im Zusammenhang mit der technischen Basis des Computers zu denken. Welche Gegebenheiten liegen in dieser vor, die bei der Entwicklung von Computerprogrammen berücksichtigt werden müssen? Welche Prozesse laufen in der Maschine ab, die die Verarbeitung einer Software regulieren? Kurzum: Wie gestaltet sich das Verhältnis zwischen der Seite der Hardware und der Seite der Software? Schon bei der bloßen Andeutung dieser Überlegungen tut sich in einem medientheoretischen Rahmen maßgeblich eine Frage von großer Bedeutung auf, welche die Gültigkeit einer Freien Software nach Stallman potentiell zunichtemachen kann: Wie frei kann eine Software eigentlich sein, die im Zusammenhang mit dem Computer begriffen wird? Und daraus folgend: Inwieweit befähigt Freie Software tatsächlich eine uneingeschränkt freie Nutzung des Computers?

Diese Fragestellung soll der folgenden Arbeit als Leitfaden dienen. Sie macht es erforderlich, zunächst zu klären, wie Freie Software in einem ‚klassischen' – d.h. Richard Stallman und der Free Software Foundation folgenden – Sinn definiert und durch welche Lizenzen sie in der Praxis nützlich gemacht wird. Mit der Vergabe eines Programms auch die Kontrolle über seine Veränderbarkeit – in Form eines offenen Quelltexts – mit zu vergeben, schließt auf der anderen Seite keineswegs aus, dass dem einstigen Urheber kein entsprechendes Urheberrecht anerkannt wird. Anschließend daran wird die Entstehung der Freien Software in einem historischen Kontext betrachtet: Welche Ereignisse gehen ihr voraus, die maßgeblichen Einfluss auf ihr Hervorbringen ausgeübt haben? Hier wird zunächst grundsätzlich die Emanzipierung der Software gegenüber der Hardware angesprochen, die in einem weiteren Schritt darin mündet, dass Software als Ware auf den Markt gelangt und demnach Machtverhältnisse zwischen den Industriefirmen auf der einen und den Nutzern auf der anderen Seite entstehen. Im Protest entwickeln sich im Laufe der 80er und 90er Jahre dann die Bewegungen der Freien Software sowie auch der *Open-Source-Software*, die es im weiteren Verlauf noch voneinander abzugrenzen gilt. Eine zentrale Rolle spielt hierbei der Hacker, welcher stets eine „Gratwanderung an den

Grenzen von Benutzerrechten, Sinn und Hardware“[6] begeht. Mit Blick auf die Geschichte der Software in der zweiten Hälfte des 20. Jahrhunderts soll schließlich hinterfragt werden, ob diese Entwicklung tatsächlich zu einer ‚Befreiung‘ des Users geführt hat. Dies soll explizit im zweiten Teil der Arbeit geklärt werden. Hier verlagert sich das Augenmerk weg von historischen Aspekten und hin zu technischen Gegebenheiten, die dem *Computing* als solchem zugrunde liegen. Es soll aufgezeigt werden, dass sich auf verschiedenen Ebenen im Computer hierarchische Verhältnisse manifestieren: Das betrifft die Programmierung, die Ausführung sowie auch die Bedienung von Software. Dass Freiheit stets auch eine Frage von Kontrolle und Macht ist, äußert sich an genau diesem Punkt, denn schließlich soll durch die Betrachtung von unterschiedlichen Mechanismen, die zwangsläufig Einschränkungen mit sich bringen, deutlich werden, dass sich im Computer eine maschinelle Macht über Handlungsmöglichkeiten äußert. Abschließend wird das Verhältnis von Software und Hardware in einen dialektischen Diskurs gestellt. Der hegelianisch geprägte Begriff der Dialektik, welcher bereits vermehrt Anwendung auf das Verhältnis von Freiheit und Kontrolle gefunden hat, soll auch hier aufgegriffen werden. All dies mündet schließlich in der Frage nach der Freiheit von Software und deren Nutzung und in möglichen Lösungsvorschlägen, indem ein neues Verständnis von Medienkompetenz propagiert wird.

Eine Arbeit wie diese kann nicht verfasst werden, ohne sich auf das Werk Friedrich Kittlers zu beziehen. Sein Standpunkt zum Thema lässt sich treffend mit den Worten Winthrop-Youngs von Software als „[…] Opium für die User, die sich weiterhin einbilden dürfen, Menschen, also Werkzeugherren oder *toolmasters* zu sein.“[7] zusammenfassen. Besondere Aufmerksamkeit soll den Aufsätzen gelten, die in *Draculas Vermächtnis* erschienen sind; wie der Zweittitel *Technische Schriften* bereits deutlich macht, propagiert Kittler hier einen maschinennahen Blick auf die Nutzung des Computers.[8] Dass in dieser Arbeit außerdem ein Werk wie Steven Levys *Hackers*, welches Kittler als ‚inhaltistisch‘ denunziert hätte, eine Betrachtung findet, soll kein Paradox darstellen, sondern vielmehr die zweigliedrige Untersuchung der Arbeit widerspiegeln.[9] Es erscheint nicht nur konsequent, sowohl die einschlägige Literatur zur

[6] Pias, Claus: „‚Children of the revolution‘. Video-Spiel-Computer als Kreuzungen der Informationsgesellschaft“, in: Ders. (Hrsg.): *Zukünfte des Computers*. Zürich/Berlin: diaphanes 2004, S. 217-240, hier S. 220.
[7] Winthrop-Young, Geoffrey: *Friedrich Kittler zur Einführung*. Hamburg: Junius 2005, S. 144.
[8] Vgl.: Kittler, Friedrich A.: *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993.
[9] Vgl.: Levy: *Hackers*.

Freien Software und zur Geschichte der Software, als auch die kritischen Werke der Medientheoretiker zu behandeln, sondern es ist auch dahingehend fruchtbar, als neue Erkenntnisse von dem einen in das andere Feld übertragen und hinterfragt werden können.

2 Freie Software

Der Ausgangspunkt dieser Arbeit soll darin bestehen, einen Blick auf das Konzept der Freie Software selbst zu werfen. Was unter dem Begriff verstanden wird und wie sich die rechtliche Handhabung gestaltet, ist deshalb Gegenstand des folgenden Kapitels.

2.1 Definition nach der *Free Software Foundation*

Um sich dem Begriff der Freien Software zu nähern, stellt sich zunächst die Frage: Welche Vorstellung von Freiheit wird impliziert, mit der sich diese Art von Software charakterisiert und insbesondere gegenüber anderen, ‚unfreien' Softwarekategorien auszeichnet? ‚Frei' kann bedeuten: autonom, eigenständig, grenzenlos, offen, aber auch verfügbar, unbesetzt oder kostenfrei. Es liegt nahe, die Antwort bei Richard Stallman zu suchen, der mit der Gründung der Free Software Foundation (FSF) eine genaue Vorstellung davon hatte, wie Freie Software – oder, genauer gesagt, im besten Fall jede vertriebene Software – aussehen sollte.[1]

Eine Definition der Freien Software fällt leichter, wenn ihr eine Klärung des Begriffs *proprietäre Software* vorangestellt wird. Schließlich besteht in dieser das nötige Gegenstück, von dem Freie Software sich abzugrenzen vermag. Als proprietär wird eine unfreie, meist kommerzielle Software beschrieben, die ihren Nutzern nur wenige Rechte einräumt, denn sie wird lediglich als maschinenlesbares Binärprogramm vertrieben, dessen Quellcode für den Nutzer nicht zugänglich ist und innerhalb der Mauern der Softwarefirmen behütet wird. „Das unfreie Programm kontrolliert die Nutzer, und der Entwickler kontrolliert das Programm. Dies macht das Programm zu einem Instrument ungerechter Macht."[2] Proprietäre Software wird unter anderem auch als *Closed Source Software* bezeichnet.[3] Im Vergleich dazu kämpft die Free Software Foundation für Benutzerfreiheiten, die jedem zustehen:

> *Freie Software* ist [eine] Software, die die Freiheit und Gemeinschaft der Nutzer respektiert. Ganz allgemein bedeutet das, dass Nutzer die Freiheit

[1] In Bezug auf ihr Motto erklärt die FSF: „Unsere Mission ist die Freiheit zu bewahren, zu schützen und zu fördern, um Rechnersoftware nutzen, untersuchen, kopieren, modifizieren und weiterverbreiten zu können und die Rechte von Freie-Software-Nutzern zu verteidigen." Free Software Foundation, Website des GNU-Systems. https://www.gnu.org/ (Abrufdatum: 03.11.2017).
[2] Free Software Foundation: „Freie Software. Was ist das?", in: https://www.gnu.org/philosophy/free-sw (Abrufdatum: 23.10.2017).
[3] Vgl.: Wichmann, Thorsten: *Linux- und Open-Source-Strategien*. Berlin/Heidelberg: Springer 2005, S. 4.

> haben[,] Software auszuführen, zu kopieren, zu verbreiten, zu untersuchen, zu ändern und zu verbessern. Freie Software ist daher eine Frage der Freiheit, nicht des Preises. Um das Konzept zu verstehen[,] sollte man an frei wie in Redefreiheit denken, nicht wie in Freibier.[4]

Insbesondere letzterer Punkt soll hervorgehoben werden, da ‚frei' häufig als ‚kostenfrei' missverstanden wird. Freie Software schließt die kommerzielle Nutzung, Entwicklung oder den kommerziellen Vertrieb aber keinesfalls aus; die Auftragsprogrammierung von Freier Software ist keine Seltenheit. Denn selbst wenn der Quelltext einer Software kostenlos verfügbar ist, können Unternehmen die Weiterentwicklung solcher Projekte mit kommerziellen Interessen vorantreiben und ihre Einnahmen dann beispielsweise durch Wartungsverträge oder Dienstleistungen erzielen.[5]

Laut Stallman fällt ein Programm in die Kategorie der Freien Software, wenn den Usern vier grundlegende Faktoren von Freiheit gewährt werden:

1. Die Freiheit, das Programm auszuführen wie man möchte, für jeden Zweck *(Freiheit 0)*.
2. Die Freiheit, die Funktionsweise des Programms zu untersuchen und eigenen Datenverarbeitungsbedürfnissen anzupassen *(Freiheit 1)*.
3. Die Freiheit, das Programm zu redistribuieren und damit seinen Mitmenschen zu helfen *(Freiheit 2)*.
4. Die Freiheit, das Programm zu verbessern und diese Verbesserungen der Öffentlichkeit freizugeben, damit die gesamte Gesellschaft davon profitiert *(Freiheit 3)*.[6]

Nur wenn all diese Forderungen in jedem möglichen Szenario erfüllt werden, handelt es sich um eine Freie Software.[7] Für einige von ihnen stellt der uneingeschränkte Zugang zum Quellcode eines Programmes eine Voraussetzung dar. Insbesondere wird in der Bewegung die Innovation durch private Rechte an einem geistigen Eigentum abgelehnt und konventionelle ökonomische Ansichten daher stark herausgefordert.

2.2 Begriffsklärungen

Häufig taucht der Begriff der *Open-Source-Software* auf, der meist synonym mit dem der Freien Software verwendet wird. Auch hier wird ein Konzept verfolgt, das Softwarenutzern mehr Rechte zugestehen will, indem frei zugänglicher Quelltext verändert und auch in abgewandelter Form weiterverbreitet werden darf. Zudem liegt das

[4] Free Software Foundation: „Freie Software".
[5] Vgl.: Wichmann: *Linux- und Open-Source-Strategien*, S. 4f.
[6] Alle Zitate: Free Software Foundation: „Freie Software".
[7] Vgl.: Ebd.

Augenmerk darauf, eine qualitativ bessere Software zu erreichen, indem viele Programmierer ihre Mitarbeit leisten und Verbesserungen am Programm anbringen können. In der Tat gibt es nur minimale Unterschiede zur Freien Software und die Übergänge zwischen den beiden Kategorien können durchaus fließend sein: „[…] sämtliche Freie Software ist öffentlich zugänglicher Quellcode und sämtliche Open-Source-Software ist beinahe frei."[8] Unterschiede lassen sich hauptsächlich in den ursprünglichen Motivationen der Gründer ausmachen. Richard Stallman grenzt die Freie Software von Open-Source-Software folgendermaßen ab:

> Die beiden Begriffe beschreiben fast die gleiche Softwarekategorie, jedoch stehen sie für Ansichten, die auf grundsätzlich verschiedenen Werten beruhen. Open Source ist eine Entwicklungsmethodik; Freie Software ist eine soziale Bewegung. Für die Freie-Software-Bewegung bedeutet freie Software eine ethisch unbedingt erforderliche, wesentliche Achtung vor der Freiheit der Nutzer. Im Gegensatz dazu betrachtet die Open-Source-Philosophie Angelegenheiten in Bezug auf wie man Software ‚besser' macht - jedoch nur im praktischen Sinne.[9]

Eine Vielzahl an Nutzern von Open-Source-Software ist dazu eingeladen, positive Netzwerkeffekte zu generieren.[10]

Zudem gibt es weitere Kategorien, die Ähnlichkeiten erwecken und eine Begriffsabgrenzung verlangen; sie nennen sich Freeware, Shareware, Demoware und Public-Domain-Software. Im Gegensatz zur Freien Software ist die *Shareware* copyright-geschützt, das heißt sie wird in der Regel mit geschlossenem Quellcode und ohne die Erlaubnis, Veränderungen vorzunehmen, veröffentlicht. Dafür geht die Weitergabe zunächst kostenfrei vonstatten - erst bei einer längerfristigen und regelmäßigen Nutzung erfolgt die Bitte um Zahlung eines bestimmten oder beliebigen Geldbetrags. Dagegen darf *Freeware* völlig gratis genutzt werden. Eine Weiterverbreitung von Kopien ist hier erlaubt, nicht aber die Modifizierung, denn auch hier ist der Quellcode nicht verfügbar. *Demoware*, auch *Crippleware* genannt, zeichnet sich durch einen eingeschränkten Leistungsumfang gegenüber einer Vollversion eines Programms aus und wird meist zum vorläufigen Test an die Nutzer herausgegeben. Unter die Kategorie der *Public-Domain-Software* schließlich fällt Software, die keinem Urheberrecht unterliegt, da es rechtlich

---

[8] Free Software Foundation: „Kategorien freier und unfreier Software", in: https://www.gnu.org/philosophy/categories.html (Abrufdatum: 23.10.2017).
[9] Stallman, Richard: „Warum Open Source das Ziel Freie Software verfehlt", in: https://www.gnu.org/philosophy/open-source-misses-the-point.de.html (Abrufdatum: 23.10. 2017).
[10] Vgl.: Osterloh, Margit; Rota, Sandra: „Open source software development. Just another case of collective invention?", in: *Research Policy*, Band 36 (2007), S. 157-171, hier S. 158f.

nicht möglich ist, sie zu schützen. Das kann etwa darauf zurückgehen, dass das Copyright verfallen ist oder der Autor gar auf ein solches verzichtet hat. Das Programm fällt damit der Gemeinheit zu, unabhängig davon, ob sein Quellcode verfügbar ist oder nicht.[11] Es fällt auf, dass sich zwischen diesen Kategorien vor allem rechtliche Unterschiede ausmachen lassen, während Freie und Open-Source-Software sich in ihrer gesellschafts-politischen Ausrichtung unterscheiden. Der Begriff der Freien Software bezieht sich stärker auf eine grundlegende Freiheit des Benutzers als die Open Source Initiative dies tut - und soll deshalb in der vorliegenden Arbeit auch bevorzugt werden.

2.3 Lizenzmodelle im Überblick

Eine oberflächliche Betrachtung des Phänomens kann durchaus zu der Annahme verleiten, Freie Software erlaube seinen Nutzern grundsätzlich alles Tun und Lassen. Jedoch machen Softwareautoren auch hier Gebrauch vom Urheberrecht, denn Software ist in Deutschland und einigen anderen Ländern rechtlich geschützt. Die Nutzungsfreiheiten werden also in Lizenzen festgeschrieben, welche als Verträge zwischen den Autoren und Nutzern fungieren. „Die Lizenzmodelle der freien Software umgehen das Urheberrecht nicht etwa oder verzichten auf seine Inanspruchnahme, sondern setzen darauf auf, um den offenen, kooperativen Prozess der Erstellung und Weiterentwicklung abzusichern."[12] Da es nämlich jedem Nutzer frei obliegt, Änderungen am Programm vorzunehmen und diese modifizierte Version an Dritte weiterzugeben, muss sichergestellt werden, dass diese Versionen nicht in proprietäre Software umgewandelt werden. So würde die ursprünglich festgeschriebene Freiheit auf dem Weg verloren gehen, was zur Folge hätte, dass einigen Endnutzern der Zugang zum Quelltext verwehrt werden würde. Die meisten Lizenzen von Freier Software folgen daher dem Konzept des *Copyleft*, welches besagt, dass alle erweiterten und modifizierten Versionen eines Programms ebenfalls als Freie Software weitergegeben werden müssen:

> Im GNU-Projekt ist unser Ziel, *allen* Nutzern die Freiheit zu gewähren, GNU-Software weiterverbreiten und ändern zu können. Wenn Mittelsmänner die Freiheit nehmen könnten[,] könnte unser Quellcode zwar ‚viele Nutzer haben', aber er würde ihnen keine Freiheit geben. […] Copyleft besagt, dass jeder, der Software - mit oder ohne Änderungen - weiterverbreitet, diese

[11] Vgl. hierzu: Grassmuck, Volker: *Freie Software. Zwischen Privat- und Gemeineigentum*. Bonn: Bundeszentrale für politische Bildung 2004, S. 278f. Sowie: Free Software Foundation: „Kategorien freier und unfreier Software".
[12] Grassmuck: *Freie Software*, S. 275.

> zusammen mit der Freiheit weitere Kopien und Änderungen machen zu dürfen übergeben muss. Copyleft garantiert, dass jeder Nutzer Freiheit hat.[13]

Entgegen dem *Copyright,*[14] durch welches proprietäre Software vor seinen Nutzern geschützt wird, garantiert das Copyleft für deren Freiheit und hat den Namen des klassischen Urheberrechts zu diesem Zweck in seiner Richtung schlichtweg umgekehrt.[15] Das Copyleft wird daher verwendet, um Quellcode und Freiheiten rechtlich untrennbar miteinander zu verknüpfen.[16]

Im Folgenden soll eine Auswahl der wichtigsten Lizenzen von Freier Software knapp dargestellt werden. Die älteste freie Quellcode-Lizenz ist die *BSD-Lizenz*. Sie wurde 1979 eigens für die Unix-Versionen erarbeitet, die von der Berkeley Universität verbreitet wurden und sowohl eigenen als auch vom mitwirkenden Telefonhersteller *AT&T* geschriebenen Code enthielten. Im Laufe der Zeit wurde eine ganze Reihe von BSD-artigen Lizenzen hervorgebracht, wie zum Beispiel *OpenBSD* oder *FreeBSD*. „Die BSD-Lizenz […] erlaubt die Verwendung und Weiterverbreitung der Software in Quell- und Binärform, mit oder ohne Veränderungen, solange der Copyright-Vermerk und der Lizenztext mitverbreitet werden […]."[17] Sie kommt ohne Copyleft aus; das Problem liegt also darin, dass es modifizierten Versionen nicht explizit vorgeschrieben wird, im Quellcode weiterverbreitet zu werden. Demnach ist es möglich, BSD-Code in proprietäre Software zu integrieren. Anders verhält es sich bei der *GNU General Public License* (GPL). Sie wurde von Richard Stallman in Zusammenarbeit mit juristischen Beratern der FSF verfasst und ist die klassische Copyleft-Lizenz, die die vier grundlegenden Freiheiten für den Nutzer umfasst. Dafür, dass das Programm auch in Zukunft bei Weitergabe frei bleibt, sorgen Bedingungen, die es verbieten, einem Nutzer diese Freiheiten vorzuenthalten sowie weitere eigene Restriktionen hinzuzufügen. Mehr als die Hälfte aller freien Softwarepakete verwendet heute die GPL.[18] Von ihr abzugrenzen ist die *GNU Lesser General Public License* (LGPL), die ebenfalls von der Free Software Foundation

---

[13] Free Software Foundation: „Copyleft. Was ist das?", in: https://www.gnu.org/licenses/copyleft.de.html (Abrufdatum: 01.11.2017).
[14] Das Copyright bezeichnet das Urheberrecht im britischen und amerikanischen Recht. Davon leitet sich auch das in Deutschland herrschende Urheberrecht ab. Grundlegend wird damit geistigem Eigentum sowohl in materieller als auch ideeller Hinsicht Schutz geboten.
[15] Stallman entnahm die Idee für den Namen von einem Briefumschlag, auf dem folgender Satz geschrieben stand: „Copyleft: all rights reversed". Vgl. hierzu: Himanen, Pekka: *Die Hacker-Ethik und der Geist des Informations-Zeitalters*. München: Riemann 2001, S. 60.
[16] Vgl.: Free Software Foundation: „Copyleft".
[17] Grassmuck: *Freie Software*, S. 279f.
[18] Vgl.: Free Software Foundation: „Lizenzen", in: https://www.gnu.org/licenses/ (Abrufdatum: 01.11.2017).

1991 entwickelt wurde. Sie lässt sich als Spezialfall einer GPL-Lizenz für Programmbibliotheken beschreiben, die als Kompromiss zwischen der GPL mit ihrem starken Copyleft und beispielsweise der BSD-Lizenz mit ihrer größeren Freizügigkeit fungiert. Im Grundlegenden werden die Freiheiten der GPL auch in der LGPL festgeschrieben; das bedeutet, dass die Bibliothek frei kopier-, modifizier- und verbreitbar sein muss, sowie dass auch Kopien und Modifikationen in ihrem Quellcode verfügbar sein müssen. Aber: „Der Hauptunterschied zur GPL besteht darin, dass Programme, die die freie Bibliothek unter dieser Lizenz einlinken und damit ein ausführliches Ganzes bilden, nicht selbst diesen Freiheiten unterstehen müssen."[19] Entwicklern wird es also erlaubt, LGPL-Programme in eigene Software zu integrieren, selbst wenn diese proprietär ist, ohne die Verpflichtung, den Quelltext der eigenen Programmteile zu veröffentlichen. Endnutzern der LGPL-Programmteile muss es allerdings möglich sein, auf diesen Zugriff zu haben und sie verändern zu können. Der Kompromiss für diese schwächere Lizenz ging mit der Strategie einher, Entwickler von proprietärer Software anlocken und ihnen Anreize für die Nutzung von Freier Software bieten zu wollen:

> Wir sind zu dem Schluss gekommen, dass schwächere Auflagen die gemeinsame Nutzung besser fördern könnten. [...] Die *Library General Public License* dient dazu, es Entwicklern unfreier Programme zu erlauben, freie Bibliotheken zu verwenden, und gleichzeitig deine Freiheit als Nutzer solcher Programme zu bewahren, die darin enthaltenen freien Bibliotheken zu verändern. [...] Es ist unsere Hoffnung, dass das zu einer schnelleren Entwicklung von freien Bibliotheken führen wird.[20]

Ein strategischer Kompromiss also, der nicht unbedingt zu großem Erfolg geführt hat; die LGPL wird heute von einzelnen, aber bei weitem nicht von allen GNU-Bibiotheken genutzt.[21]

[19] Grassmuck: *Freie Software*, S. 290.
[20] Hierbei handelt es sich um die Präambel zur LGPL. Zitiert nach: Ebd., S. 291.
[21] Vgl.: Free Software Foundation: „Lizenzen".

3 Historische Einordnung

„Eine Geschichte der Softwareentwicklung zu schreiben[,] ist allein deshalb ein Problem, weil dabei die Entscheidung getroffen werden muss, ab wann von Software gesprochen werden kann."[1]

3.1 Frühe Softwaregeschichte

Tatsächlich kann nicht klar bestimmt werden, welcher Zeitpunkt in der Geschichte als der Beginn einer Entwicklung von Software als solcher festgelegt werden kann; hierzu variieren die Meinungen in der Forschung.[2] Eine erste Verwendung des Begriffs im heutigen Sinn lässt sich 1958 in einem Aufsatz John W. Tukeys zur Konkreten Mathematik finden: „Today the ‚software' comprising the carefully planned interpretive routines, compilers, and other aspects of automative programming are at least as important to the modern electronic calculator as its ‚hardware' of tubes, transistors, wires, tapes and the like."[3] Damit wird Software erstmalig als autonomer Part in der Rechnerkomposition wahrgenommen und als solcher hervorgehoben. Zuvor wurde sie als untrennbarer Teil der Hardware, als Einheit mit all den Röhren, Transistoren, Drähten und Bändern und in keiner Weise als unabhängig Existierendes verstanden. Unternehmen wie IBM, das die Computerindustrie seit Beginn der 1950er Jahren dominierte, verdienten ihr Geld zu Zeiten Tukeys noch ausschließlich mit dem Verkauf von Hardware; Systemsoftware wurde zusammen mit der Computerhardware als Teil des Lieferumfangs vertrieben. Je nach Bedarf entwickelten Nutzer ihre eigenen Anwendungsprogramme und tauschten diese untereinander aus. Die Quelltexte der Programme waren demnach stets frei und im weitesten Sinne kostenlos. Dies bedeutete auch einen großen Aufwand, denn die Firmen, die sich eine der kolossalen Rechneranlagen zulegten, mussten eine Menge an Personal für die Programmierung einstellen, um den Einsatz der Maschine bewältigen zu können.[4]

---

[1] Korb, Joachim: „Geschichte der Softwareprogrammierung: ‚Freie Software für Freiheit und Gerechtigkeit'", in: *Perspektive'89*, 21.12.2006. http://perspektive89.com/2006/12/21/geschichte_der_softwareprogrammierung_freie_software_fur_freiheit_und_gerechtigkeit (Abrufdatum: 05.11.2017).

[2] Aufgrund der Ausrichtung der vorliegenden Literatur bezieht sich die folgende Abhandlung größtenteils auf die Entwicklungen in den USA.

[3] Tukey, John W.: „The Teaching of Concrete Mathematics", in: *The American Mathematical Monthly*, Band 65, Heft 1 (1958), S. 1-9, hier S. 2.
Auch Paul Ceruzzi bestätigt diesen Zeitpunkt in etwa, ohne dass er dies allerdings weiter ausführt oder begründet: „The term [software] seems to have come into use around 1959." Ceruzzi: *A history of modern computing*, S. 353, Anmerkung 22.

[4] Vgl.: Ebd., S. 8f.

Nach Ceruzzi beginnt die Softwareentwicklung bereits im Jahr 1944 an der US-amerikanischen Maschine *Mark I*.[5] Dieser Rechner, von Howard Aiken entwickelt und aus elektromechanischen Bauteilen gebaut, konnte mit einem Lochstreifen mit 24 Spuren pro Anweisung programmiert werden. Grace Murray Hopper wurde an der Havard Universität mit seiner Programmierung beauftragt, während Aiken stets für den technischen Part der Maschine zuständig war. Somit waren die Bereiche der Soft- und Hardware arbeitstechnisch klar voneinander abgetrennt. Daraus folgt: „Thus began the practice of computer programming in the United States."[6] Theoretisch war es möglich, den Mark I zu jedem Zweck frei zu instruieren, allerdings war er, wie die meisten frühen Rechenmaschinen, mit militärischer Intention gebaut worden und wurde somit hauptsächlich für die Berechnung von Feuer- und Flugtabellen für die US-amerikanische Marine genutzt.[7]

Je leistungsfähiger die Rechner im Laufe der Jahre wurden, desto anspruchsvoller wurden auch die Aufgaben, die man ihnen übertrug. Es wurde erforderlich, immer mehr Programme zu entwickeln, die alle nötigen Verwaltungsaufgaben übernehmen konnten. Zudem stellte es beim Rechnerwechsel von einer Generation zur nächsten für die Nutzer oft ein Problem dar, die selbst geschriebene Software auf die Anforderungen der neuen Maschine abzustimmen. Aus diesen Gründen trat 1955 die Benutzergruppe *SHARE* zutage, die es sich zunächst zur Aufgabe gemacht hatte, die Software des *IBM 701* auf den nachfolgenden *IBM 704* anzupassen, sich mit der Zeit – stetigem Wachstum der Industrie sei Dank – aber einer umfassenden Programmbibliothek widmete. Bei der Gemeinschaft aus Mitarbeitern der verschiedensten Firmen rund um Los Angeles ging es in erster Linie um einen Erfahrungsaustausch.[8] Gewissermaßen lässt sich hier schon der Kollaborationsgedanke finden, den Stallman Jahrzehnte später in der Bewegung der Freien Software verfechtet. Es lässt sich festhalten, dass sich die Trennung der Arbeitsbereiche von Hardware und Software in den 50er Jahren durchgesetzt und damit auch die Bedeutung der Software, „something that, by definition, has no physical essence,

[5] Der Verständlichkeit wegen wird für die weiteren Ausführungen der Begriff der Software verwendet, auch wenn dieser zu den genannten Zeiten noch nicht im Gebrauch war.
[6] Ceruzzi: *A history of modern computing*, S. 82.
[7] Vgl.: Ebd., S. 81f.
[8] Während einige daran festhalten, dass dem Namen *SHARE* kein tieferer Sinn zugrunde liegt, gibt es andere Meinungen, die behaupten, es handele es sich dabei um eine Abkürzung für „Society to Help Avoid Redundant Effort". Vgl. hierzu: Ebd., S. 368, Anmerkung 32.

precisely that which is not ‚hardware'"[9], als unabdingbaren Zusatz zum Computer verbreitet hat.

Es dauerte nicht lange, bis sich Computerhersteller darüber bewusst wurden, dass aus weit verbreiteter Software ein entsprechender Wettbewerbsvorteil hervorging, den es zu sichern galt. In den 60er Jahren kamen daher erstmals Diskussionen zum rechtlichen Schutz von Software auf: Einerseits wurde hier über eine Neufassung des US-amerikanischen Copyrights verhandelt, sodass die Regelungen auch auf Computerprogramme anwendbar seien. Da ein Programm beim Gebrauch im Computer allerdings in den Binärcode umgewandelt werden muss und daher eine neue Form erlangt, wurde Software andererseits nicht als Text im eigentlichen Sinne verstanden und konnte deshalb auch nicht so einfach unter den Schutz des Copyrights fallen. Deshalb wurde alternativ die Übertragbarkeit des Patentrechts auf Software diskutiert, um diese rechtlich sichern zu können.[10]

Zum Ende der 1960er Jahre hin änderten sich die Rahmenbedingungen für die kommende Produktion von Computerprogrammen grundlegend. Im Dezember 1968 gab IBM bekannt, dass seine Hardware und Software künftig nicht mehr gebündelt in Paketen, sondern nur noch getrennt voneinander verkauft werden würden. Bei seinem Vorgehen wurde das Unternehmen, das den Computermarkt weit vor allen anderen Herstellerfirmen beherrschte, stark von der US-amerikanischen Regierung unter Druck gesetzt. Der Markt sollte auch für andere Firmen zugänglich werden und man erhoffte sich, dies mithilfe des Verkaufs der Softwarekomponente zu erreichen. Mit der Markteinführung des *System/360* schaffte IBM bereits 1964 einen Wegbereiter hierfür. Es handelte sich dabei erstmals um eine Computerreihe, die austauschbare Software benutzen konnte. Dies bedeutete für den Kunden eine Investition, bei der er auf Dauer sparen konnte, steigerte aber gleichzeitig den Wert der kompatiblen Software.[11] Der Prozess des sogenannten *Unbundlings* kann insofern als Revolution angesehen werden, als Software erstmals zum eigenständigen Produkt und ein völlig neues Absatzgebiet geschaffen wurde: „Ein neuer Markt entstand, der im Laufe der Zeit den eigentlichen Computermarkt an Kapitalkraft

[9] Ceruzzi: *A history of modern computing*, S. 79.
[10] Dieser Sachverhalt kann bis heute nicht geklärt werden und ruft immer wieder Schwierigkeiten hervor. Vgl. dazu auch: Korb: „Geschichte der Softwareprogrammierung".
[11] Vgl.: Ceruzzi: *A history of modern computing*, S. 106 und S. 144f.

bei weitem übersteigen sollte.“[12], so Joachim Korb. „Unbundling gave birth to the multibillion-dollar software and services industries, of which IBM is today a world leader.“[13], heißt es auf der Website IBMs. Mit der Öffnung des Softwaremarktes war der Weg für proprietäre Software gesichert: „It led to a commercial software industry that needed to produce reliable software in order to survive. […] Software came of age in 1968.“[14]

Ein Problem, das zu dieser Zeit auftrat und schließlich zur sogenannten *Softwarekrise* führte, war es jedoch, dass die Produktionskosten für Software die für Hardware allmählich überstiegen. Mit der zunehmenden Komplexität von Software konnten die bisher genutzten Techniken nicht mehr mithalten. Das Problem wurde von Edsger W. Dijkstra anlässlich des Turing Awards 1972 wie folgt zusammengefasst:

> […] the major cause is … that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. […] the electronic industry has not solved a single problem, […] it has created the problem of using its products.[15]

Öffentlich anerkannt wurde die Krise bereits 1968 während einer Nato-Konferenz in Garmisch-Partenkirchen, welche weithin für großes Aufsehen sorgte.[16] Es wurde deutlich, „dass individuelle Ansätze zur Programmentwicklung sich nicht auf große und komplexe Softwaresysteme übertragen ließen“[17]. Für Dijkstra lag die Lösung darin, die Softwareprogrammierung effektiver zu gestalten:

> If software development were to continue to be the same clumsy and expensive process as it is now, things would get completely out of balance. […] therefore we *must* learn to program an order of magnitude more effectively. To put it in another way: as long as machines were the largest item on the budget, the programming profession could get away with its clumsy techniques, but that umbrella will fold very rapidly.[18]

So wurde auf der Konferenz zum ersten Mal der Begriff des *Software Engineering* vorgeschlagen, welcher besagt, dass bei der Softwareentwicklung eine ingenieurmäßige

---

[12] Korb: „Geschichte der Softwareprogrammierung“.
[13] IBM Corporation: „Chronological History of IBM. 1960s“, in: IBM Archives Online. http://www-03.ibm.com/ibm/history/history/decade_1960.html (Abrufdatum: 20.11. 2017).
[14] Ceruzzi: *A history of modern computing*, S. 108.
[15] Dijkstra, Edsger W.: „The Humble Programmer“, in: *Communications of the ACM*, Band 15, Heft 10 (1972), S. 859-866, hier S. 861.
[16] Vgl.: Ebd., S. 863.
[17] Sommerville, Ian: *Software Engineering*. München: Pearson 2012, S. 29.
[18] Dijkstra: „The Humble Programmer“, S. 863.

Vorgehensweise verfolgt werden sollte. In den kommenden Jahren wurde eine Menge dementsprechender Methoden und Techniken hervorgebracht, darunter insbesondere die strukturierte Programmierung.[19]

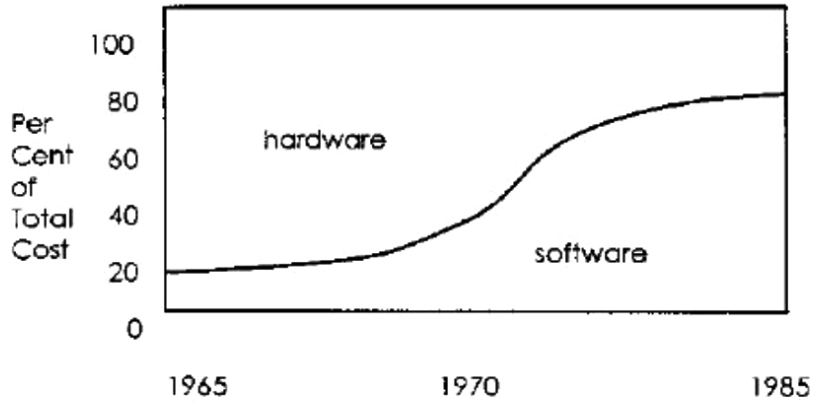

Abb. 3.1: Eine Gegenüberstellung der anteiligen Kosten von Hardware und Software innerhalb eines zeitlichen Rahmens von 1965 bis 1985.

3.2 Entstehung der Hackerbewegung

Die Computerindustrie sollte sich im kommenden Jahrzehnt grundlegend wandeln: Der für Firmen produzierte Mainframecomputer erfährt als vorherrschendes Produkt dieser Industrie Ende der 70er Jahre eine Ablösung durch den für den Privatnutzer bestimmten Heimcomputer. Einen entscheidenden Anteil hieran trägt die Subkultur der Hacker, welche von einer derartigen Faszination durch den Computer geprägt war, dass ein Großteil der Innovationen in diesem Bereich ihrem Einsatz zuzuschreiben ist.

Als maßgebende Quelle dient Steven Levys umfassende Abhandlung einer Hackerkultur der frühen 60er, 70er und 80er Jahre, die er mit den folgenden Worten einleitet:

> Though some in the field used the term ‚hacker' as a form of derision, implying that hackers were either nerdy social outcasts or ‚unprofessional' programmers who wrote dirty, ‚nonstandard' computer code, I found them quite different. Beneath their often unimposing exteriors, they were adventurers, visionaries, risk-takers, artists… and the ones who most clearly saw why the computer was a truly revolutionary tool.[20]

Der Akt des *Hackens* hat seinen Ursprung im universitären Umfeld: Nach Levy wird dieser gegen Ende 1958 am Massachusetts Institute of Technology (MIT) in Cambridge in einem Studentenclub für Modelleisenbahnbau geboren, der sich Tech Model Railroad

[19] Vgl.: Sommerville: *Software Engineering*, S. 29.
[20] Levy: *Hackers*, S. ix.

Club (TMRC) nannte. Innerhalb des Clubs gab es zwei Gruppierungen - einerseits diejenigen, die sich für die Gestaltung der Modellbahnen interessierten, und andererseits das Signals and Power Subcommittee (S&P), das für die Elektrotechnik zuständig war, um die Bahnen zum Laufen zu bringen. Hierzu gehörten unter anderem die Studenten Peter Samson, Bob Saunders und Alan Kotok, die sich zunehmend auch für Computerprogrammierung interessierten und sich bald regelmäßig Zugang zu den universitätseigenen Computern verschafften. Die technikaffinen Studenten pflegten sich in einer Reihe von Codewörtern untereinander zu verständigen, darunter auch der ‚hack', „[…] a project undertaken or a product built not solely to fulfill some constructive goal, but with some wild pleasure taken in mere involvement […]."[21] Hinzu kommt: „[…] to qualify as a hack, the feat must be imbued with innovation, style, and technical virtuosity."[22] Somit hatten die Mitglieder des TMRC sich selbst erstmals als Hacker bezeichnet.

Zu der Zeit befand sich am MIT ein *TX-0*, einer der frühesten transistorbasierten Computer, der direkt über einen Lochstreifenleser programmiert werden konnte und seinen Nutzer über einen Bildschirm, Lautsprecher und eine Reihe kleiner Lampen über den Status des Programmes auf dem Laufenden hielt.[23] Im Gegensatz zu dem zuvor verwendeten IBM 704, der stets von einer regelrechten Mannschaft von professionellen Bedienern bewacht wurde, war der TX-0 erstmals leichter zugänglich. Die Hacker des TMRC machten sich bald daran, die Technik der Maschine genau kennenzulernen und den Möglichkeiten nachzugehen, die ihnen nun offenstanden. „Something new was coalescing around the TX-0: a new way of life, with a philosophy, an ethic, and a dream."[24] Aus der zunehmenden Symbiose zwischen Mensch und Maschine heraus etablierte sich eine Hackerethik, die mehr stillschweigend vereinbart als diskutiert oder in einem Manifest aufgezeichnet wurde. Levy fasst die Grundsätze, denen die Hacker am MIT folgten, in sechs Punkten zusammen:

1. Access to computers - and anything which might teach you something about the way the world works - should be unlimited and total. Always yield to the Hands-On Imperative!
2. All information should be free.
3. Mistrust Authority - Promote Decentralization.

[21] Levy: *Hackers*, S. 9.
[22] Ebd., S. 9f.
[23] Ceruzzi schreibt hierzu: „It was the culmination of ideas about interactive computing that began with the TX-0 at MIT." Ceruzzi: *A history of modern computing*, S. 271.
[24] Levy: *Hackers*, S. 26.

4. Hackers should be judged by their hacking, not bogus criteria such as degrees, age, race, or position.
5. You can create art and beauty on a computer.
6. Computers can change your life for the better.[25]

Zentral ist hierbei insbesondere die uneingeschränkte Nutzung der Rechner. Auch die auf den Lochstreifen geschriebenen Programme wurden stets für andere Nutzer offen liegen gelassen, sodass auch auf diese frei zugegriffen und Verbesserungen vorgenommen werden konnten. Die Technik sowie alle Informationen über die Technik wurden als kollektives Gut angesehen. Denn: „What was a computer but something which benefited from a free flow of information?“[26] Man sah zurückgehaltene Informationen als Schranke an, die es verhinderte, das Potential der Maschine vollends auszunutzen zu können. Demnach galt auch die Entwicklung der verschiedensten Computerprogramme eher als Mittel zum Zweck: „It was common, if you wanted to do a task on a machine and the machine didn't have the software to do it, to write the proper software so you *could* do it.“[27]

Mit dem *PDP-1*, dem direkten Nachfolger des TX-0 auf dem Campus, wurde dem MIT 1961 erstmals ein Minicomputer bereitgestellt. An diesem gelang es einem Studenten namens Steve Russell erstmals, einen Display Hack durchzuführen: Er programmierte das 2D-Spiel *Spacewar!*, das den Kampf zweier Raumschiffe auf der Bildschirmfläche simulierte. Das Spiel wurde jedem frei zur Verfügung gestellt. Ein weiterer Meilenstein war das Projekt *MAC*, das, dem Konzept des *Time-Sharing* folgend, 1961 am Tech Square des MIT beheimatet wurde. Anfangs war ein Großteil der Hacker dem Time-Sharing-System eher abgeneigt, da dieses sehr langsam war und nicht der Eindruck entstand, als könne man das System in seinem vollen Umfang nutzen. Die beiden Studenten Richard Greenblatt und Stewart Nelson widmeten sich daher der Programmierung eines neuen Systems, dem sogenannten *Incompatible Time-sharing System*. Dieses ermöglichte allen Nutzern einen vollständigen Zugriff auf alle Daten eines Systems und erlaubte es zudem, viele Programme gleichzeitig auszuführen. „The idea was that computer programs belonged not to individuals, but to the world of users.“[28] Das System verdeutlicht die Priorisierung des Kollektivs vor dem Individuum, welche für die Zeit der frühen Hacker charakteristisch war.

---

[25] Alle Zitate: Levy: *Hackers*, S. 27-33.
[26] Ebd., S. 28.
[27] Ebd., S. 60.
[28] Ebd., S. 115.

Im Laufe der 60er Jahre, mit der stärkeren Verbreitung des Computers, kam auch die Hackerkultur in Umlauf. So kann etwa Stanford's Artificial Intelligence Laboratory (SAIL) oder die Carnegie-Mellon-Universität (CMU) aufgeführt werden, die sich beide als wichtige Zentren für Computerwissenschaften und die Forschung der Künstlichen Intelligenz erwiesen.[29] Auch außerhalb des universitären Umfelds fanden die Prinzipien der Hacker in Firmen und Organisationen mehr und mehr Anklang. Als wichtige Etappe ist insbesondere das Projekt *Community Memory* hervorzuheben, welches 1973 in Berkeley, Kalifornien, aufkam und für Levy den Start einer sogenannten ‚zweiten Hacker-Generation' markiert, geprägt durch „a type of hacker who not only lived by the Hacker Ethic but saw a need to spread that gospel as widely as possible."[30] Zentraler Initiator Lee Felsenstein wollte mit Community Memory ein Kommunikationssystem etablieren, das einen dezentralisierten Informationsfluss zwischen unabhängigen, anonymen Individuen unterstützte. In einem Schallplattenladen der Universität Berkeley wurde zu diesem Zweck ein Computerterminal installiert, das als öffentliches elektronisches Schwarzes Brett fungierte, denn es war über eine Telefonleitung mit einem Rechner verbunden, der auf die Befehle ‚ADD' und ‚FIND' hin Nachrichten veröffentlichen und finden konnte und den Informationsaustausch innerhalb des Kundenkreises so effizienter machte. Genutzt wurde es als Marktplatz für gebrauchte Platten, als Plattform für Kritiken, aber auch für allgemeinere Anliegen wie etwa Wohnungsgesuche, und erfüllte damit seinen Zweck, das Leben der Nutzer zu erleichtern.[31] Gerade die direkte Zugänglichkeit spielte hierbei eine große Rolle: „It became clear that the crucial element was the fact that people could walk up to the terminals and use them hands-on, with no one else interposing their judgement. The computer system was not interposing itself between the individuals who used it, either."[32]

Mit der Entwicklung und Kommerzialisierung des Mikroprozessors durch *Intel* erfuhr die Computertechnologie 1971 eine Revolution, die sich auch auf das Treiben der Hacker auswirkte.[33] Die ersehnte Verbreitung des Computers, einhergehend mit einer Abkehr von

---

[29] Vgl.: Raymond, Eric S.: *The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol: O'Reilly 1999, S. 10.
[30] Levy: *Hackers*, S. 144.
[31] Vgl.: Ebd., S. 155ff. Sowie: Höltgen, Stefan: „‚All Watched Over by Machines of Loving Grace'. Öffentliche Erinnerungen, demokratische Informationen und restriktive Technologien am Beispiel der ‚Community Memory'", in: Reichert, Ramón (Hrsg.): *Big Data. Analysen zum digitalen Wandel von Wissen, Macht und Ökonomie*. Bielefeld: transcript 2014, S. 385-403, hier S. 387f.
[32] Lee Felstenstein, 1993, zitiert nach: Höltgen: „All Watched Over by Machines of Loving Grace", S. 397.
[33] Faggin, Federico: „How we made the microprocessor", in: *Nature Electronics*, 08.01.2018. https://www.nature.com/articles/s41928-017-0014-8 (Abrufdatum: 29.01.2018).

den großen Rechenanlagen,[34] konnte dadurch vorangetrieben werden: „[…] small and powerful computers in great number could really change the world."[35] Bis der Mikroprozessor den Privatsektor erreichte, sollte es allerdings noch wenige Jahre dauern: Im Januar 1975 erschien der *Altair 8800*, einer der ersten Heimcomputer, gebaut um Intels 8080-Mikroprozessor von der Firma *MITS*. Sein Erfinder Ed Roberts hatte es sich zur Aufgabe gemacht, einen Computer für die Massen zu erschaffen. Intels Mikroprozessor bot ihm diese Möglichkeit, sodass er den Heimcomputer zu einem verhältnismäßig günstigen Preis anbieten konnte und dieser für weitaus mehrere potentielle Nutzer erschwinglich wurde - „a computer to the world"[36] sozusagen. Wenn der Altair in seiner Basisversion nur sehr eingeschränkt nutzbar war - er verfügte lediglich über ein Paneel von Schaltern und Leuchtdioden an der Frontseite; ein Display oder eine Bedienungstastatur gab es noch nicht - so war er für die Hacker doch produktiv nutzbar: „It was a computer, and what hackers could do with it would be limited only be their own imaginations."[37]

Mehr Fortschritt brachte der sogenannte *Homebrew Computer Club*, dessen erste Sitzung im Mai desselben Jahres stattfand. Er lässt sich als Verein für Computerenthusiasten beschreiben, die stets das Ziel vor Augen hatten, Computer für alle Menschen zugänglicher zu machen. Der Homebrew Computer Club ist schließlich der Ort, an dem sich das manifestiert, was als ‚Hardware Hacking' bezeichnet werden kann:[38] „These were people intensely interested in getting computers into their homes to study, to play with, to create with… and the fact that they would have to build the computers was no deterrent."[39] Zu seinen Mitgliedern zählten in der Anfangszeit Gordon French, Fred Moore, Lee Felsenstein, Bob Albrecht, Steve Wozniak und einige mehr. Kurzum:

> The people in Homebrew were a mélange of professionals too passionate to leave computing at their jobs, amateurs transfixed by the possibilities of technology, and techno-cultural guerrillas devoted to overthrowing an oppressive society in which government, business, and especially IBM had

[34] Die Untauglichkeit der Großrechenanlagen liegt aus vielen Gründen auf der Hand: Als Beispiel soll Levys Beschreibung des IBM 704, auch ‚The Hulking Giant' genannt, dienen: „The IBM 704 cost several million dollars, took up an entire room, needed constant attention from a cadre of professional machine operators, and required special air-conditioning so that the glowing vacuum tubes inside it would not heat up to data-destroying temperatures." Levy: *Hackers*, S. 5.
[35] Ebd., S. 144.
[36] Ebd., S. 182.
[37] Ebd., S. 185.
[38] Vgl.: Ebd., S. x. Diese von Levy gewählte Kategorisierung trifft insofern zu, als die Hacker sich zu dieser Zeit konkret mit der Herstellung von Geräten beschäftigt haben. Dennoch war es seit Beginn des Hacker-Daseins ihre oberste Prämisse, einen freie Zugänglichkeit zur Hardware zu erlangen.
[39] Ebd., S. 196.

regulated computers to a despised Priesthood.[40]

Sie trafen sich in regelmäßigen Abständen, um Informationen auszutauschen, sich gegenseitig Tipps zu geben und über technische Neuigkeiten auf dem Laufenden zu sein. Vorangetrieben durch den regen kreativen Austausch entstand so in kurzer Zeit eine Reihe technischer Innovationen; darunter diverse Add-On Boards für den Altair 8800, die von Jim Warren herausgegebene Computerzeitschrift *Dr. Dobbs Journal* (1976) und vollständige Heimcomputer wie der *IMSAI* (1976) oder der *Sol-20* (1977). Zur gleichen Zeit entstehen auch Softwareprojekte, die als Freie Software eingestuft werden können, einige Jahre noch vor dem Zustandekommen der Free Software Foundation; so etwa das von Dennis Allison entwickelte *Tiny BASIC* (1975) und Tom Pittmans Interpreter für Tiny BASIC (1976).[41] Mit dem von Steve Wozniak entworfenen und 1977 veröffentlichten *Apple II* schließlich gelingt der Durchbruch des Heimcomputers: „It was Wozniak and the computer he'd design[ed] that would take the Hacker Ethic, at least in terms of hardware hacking, to its apogee."[42] Es war eine Maschine geschaffen, die nicht nur für technikaffine Hobbybastler, sondern auch für die Mehrheit der Menschen einen Nutzen darstellen konnte. Dies stellt für Levy sowohl den Höhepunkt der Hacker Ethik, als auch eine erste Vermischung der Hacker- und der Industriewelt dar, mit dem Ergebnis, dass das Produkt des Computers erstmals eine weite Verbreitung erlangt.[43]

3.3 Von proprietärer zu Freier Software

„[…] after the Apple II and its floppy drive were available, one could say that hardware advances no longer drove the history of computing […]."[44] Dies hatte bereits die Softwarekrise deutlich gemacht und so soll im folgenden Abschnitt verstärkt der Softwaregeschichte Aufmerksamkeit gezollt werden. Nach den Maßnahmen des Unbundlings durch IBM hatte sich proprietäre Software im Laufe der 70er Jahre in der Informationsindustrie durchgesetzt und als Standard etabliert. Das sollte sich jedoch im Jahr 1984 ändern – dem Zeitpunkt nämlich, als ein „freies Paralleluniversum zur proprietären Software"[45] entsteht.

---

[40] Levy: *Hackers*, S. 200.
[41] Vgl.: Ebd., S. 226ff.
[42] Ebd., S. 240.
[43] Vgl.: Ebd., S. 255ff.
[44] Ceruzzi: *A history of modern computing*, S. 268.
[45] Grassmuck: *Freie Software*, S. 285.

Erheblichen Einfluss auf die Entstehung der Freien Software hatte das von Ken Thompson und Dennis Ritchie entwickelte Betriebssystem *UNIX*. In seinen Anfängen nach 1969 war es als internes System für *Bell Laboratories* entwickelt worden, erlangte aber schon nach kürzester Zeit auch außerhalb des Unternehmens Bekanntheit. Insbesondere an Universitäten wurde das System für nur geringe Summen samt seines Quelltextes herausgegeben. Damit wurde es einigen Nutzern ermöglicht, Veränderungen am Programm vorzunehmen, wenn ihnen eine Anpassung an die eigenen Arbeitsbedürfnisse oder eine grundlegende Verbesserung der Fähigkeiten notwendig erschien. Auch Thompson und Ritchie konnten ihren Profit aus dieser Entscheidung ziehen. So äußerten sie sich 1978 in *The Bell System Technical Journal*:

> If designers of a system are forced to use that system, they quickly become aware of its functional and superficial deficiencies and are strongly motivated to correct them before it is too late. Because all source programs were always available and easily modified on-line, we were willing to revise and rewrite the system and its software when new ideas were invented, discovered, or suggested by others.[46]

Mit diesem Maß an Freiheit, das mit der Vergabe von UNIX an seine User übertragen wurde, steht das Betriebssystem in starkem Kontrast zu einem Großteil der vertriebenen Software in dieser Zeit:[47] „[…] most computer vendors guarded source code as their family jewels, seldom gave it out, and did all they could to lock a customer into their products.“[48] Trotzdem bleibt festzuhalten, dass es sich bei UNIX immer noch um eine proprietäre Software handelte. Denn die Entscheidung, Nutzern den Zugriff auf den Quelltext zu gewähren, oblag immer noch den Eigentümern und konnte jederzeit durch das Entziehen der entsprechenden Rechte rückgängig gemacht werden. Problematisch war es zudem, dass die Nutzung von UNIX durch Betriebsgeheimnisse und kommerzielle Interessen eingeschränkt war.[49]

Um den Machtverhältnissen zwischen der Softwareindustrie und den Usern entgegenzuwirken, rief Richard Stallman 1983 schließlich das sogenannte *GNU-Projekt*[50] ins Leben:

---

[46] Ritchie, Dennis M.; Thompson, Ken: „The UNIX Time-Sharing System“, in: *The Bell System Technical Journal*, Band 57, Heft 6 (1978), S. 1905-1929, hier S. 1927.
[47] Vgl.: Ceruzzi: *A history of modern computing*, S. 282ff.
[48] Ebd., S. 283.
[49] Vgl.: Raymond: *The Cathedral and the Bazaar*, S. 62.
[50] Der Name *GNU* steht für „GNU's Not Unix“, wurde aber auch aufgrund seines tatsächlichen Vorkommens im englischen (und weiteren) Wortschätzen sowie seiner belustigenden Aussprache gewählt. Vgl. dazu: Free Software Foundation: „Geschichte des GNU-Systems“, in: https://www.gnu.org/gnu/gnu-history (Abrufdatum: 05.01.2018).

> Das GNU-Projekt entstand im Jahr 1983 als eine Möglichkeit, den zusammenarbeitenden Geist zurückzubringen, der in früheren Tagen in der Rechnerwelt vorherrschte – um von Eigentümern proprietärer Software auferlegte Hindernisse zu beseitigen, die die Zusammenarbeit verhinderte und dadurch [eine solche] wieder möglich werden zu lassen.[51]

Stallman war seit 1971 Mitarbeiter am Labor für künstliche Intelligenz am MIT gewesen, wo er Zeuge der Hacker Ethik wurde und sich zunehmend selbst an dieser orientierte. So schrieb er etwa den Texteditor *EMACS*, welchen er 1976 kostenlos veröffentliche – unter der Bedingung, dass das Programm stets mit allen neu hinzugekommenen Erweiterungen und Verbesserungen weitergegeben werden sollte. Im Laufe der 70er Jahre sah Stallman sich jedoch wachsenden Schranken konfrontiert, wie zum Beispiel einer neuen Sicherheitspolitik am Labor, die vorschrieb, dass künftig nur noch mit einem Passwort Zugriff auf ein System gewährt wurde.[52] Einhergehend damit manifestierte sich bei ihm der Gedanke, dass Computerprogramme immer frei sein sollten: „I don't believe that software should be owned. Because [the practice] sabotages humanity as a whole. It prevents people from getting the maximum benefit out of the program's existence."[53] Die deutlich sichtbare Kommerzialisierung des Computers mit ihren Sicherheits-anforderungen und bürokratischen Prozessen hielt auch Einzug ins MIT und führte dazu, dass sich die Seiten dort zunehmend spalteten. Immer mehr Nutzer konnten die Maschinen nicht mehr selbst reparieren oder ihre Software anpassen und wandten sich daraufhin den herstellenden Firmen zu. Im Protest dagegen und als logische Konsequenz daraus verließ Stallman die Abteilung für Künstliche Intelligenz und verschrieb sich dem GNU-Projekt, um die Hacker Ethik, die nicht mehr weiter in ihrer unverfälschten Form existieren konnte, welche sie zu früheren Zeiten am MIT erlangt hatte, in die Außenwelt zu tragen.[54]

Im September 1983 kündigte Stallman in einem Beitrag im Internetforum *Usenet*, kurz für *Unix User Network*, an: „Free Unix! Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU […], and give it away free to everyone who can use it. Contributions of time, money, programs and equipment are greatly needed."[55] Dem fügt er später noch hinzu: „So that I can continue to use computers

---

[51] Free Software Foundation: „Geschichte des GNU-Systems".
[52] Vgl.: Levy: *Hackers*, S. 421ff.
[53] Richard Stallman, 1983, zitiert nach: Ebd., S. 425.
[54] Vgl.: Ebd., S. 423ff.
[55] Richard Stallman, 1983, zitiert nach: Free Software Foundation: „Initial Announcement", in: https://www.gnu.org/gnu/initial-announcement.en.html (Abrufdatum: 05.01.2018).

without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free."[56] Da es die Voraussetzung für die Nutzung eines Rechners darstellt, stand für Stallman die Entwicklung eines freien Betriebssystems als erster Punkt auf der Agenda. UNIX hatte sich als Betriebssystem bereits bewährt und eine weite Verbreitung gefunden, sodass Stallman sich dazu entschied, sein eigenes Betriebssystem mit UNIX kompatibel zu machen. Dies sollte den Umstieg für die Nutzer möglichst einfach gestalten. Für die Umsetzung wurden eine Menge Gelder benötigt, sodass Stallman im Jahr 1985 die Free Software Foundation gründete, um entsprechende Finanzmittel zusammentragen zu können. Von 1984 bis 1990 wurden – mit Ausnahme des Betriebssystemkerns – alle wichtigen Komponenten geschrieben.[57] Einen großen Teil der Programmierarbeiten erledigte Stallman selbst; er erhielt zudem Hilfe von einigen Mitentwicklern. Zusammen bauten sie eine ganze Reihe von Softwarekomponenten auf, für die es im Unternehmen weitaus größere Teams benötigt hätte, so Ceruzzi.[58] Seine Vollendung fand GNU schließlich im Jahr 1992, als es um den Kernel *Linux* ergänzt wurde.[59] Dieser wurde – zunächst unabhängig von den Bestrebungen Stallmans – von Linus Torvalds entwickelt. Unzufrieden mit der UNIX Version *Minix*, welche dieser benutzt hatte, nahm er sich vor, eine eigene Version für seinen IBM kompatiblen PC zu schreiben, die all seinen Anforderungen gerecht werden würde.[60] Zunächst nahm er sich der Entwicklung alleine an, bezog im Laufe der Zeit jedoch mehr und mehr Mitentwickler in das Projekt ein. Letztendlich lässt sich festhalten, dass Linux entwickelt wurde „by several thousand developers scattered all over the planet, connected only by the tenuous strand of the Internet"[61].

---

[56] Richard Stallman, 1983, zitiert nach: Free Software Foundation: „Initial Announcement".
[57] Vgl.: Free Software Foundation: „Geschichte des GNU-Systems".
Stallman wird allerdings nicht müde, die User daran zu erinnern, dass das vollständige Betriebssystem nicht bloß Linux, sondern vielmehr *GNU/Linux* genannt werden sollte. Hierzu hat sich ein regelrechter Namensstreit entwickelt; Vgl. dazu zum Beispiel: Ceruzzi: *A history of modern computing*, S. 341. Oder: Free Software Foundation: „GNU/Linux: Häufig gestellte Fragen. Warum wird das benutzte System GNU/Linux genannt, nicht ‚Linux'?", in: https://www.gnu.org/gnu/gnu-linux-faq.de.html#why (Abrufdatum: 05.01.2018).
[58] Vgl.: Ceruzzi: *A history of modern computing*, S. 340.
[59] Vgl.: Free Software Foundation: „Geschichte des GNU-Systems".
[60] Auch Torvalds kündigte sein Vorhaben in einem Internetforum an; dort schrieb er: „I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386 (486) cones… I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat." Zitiert nach: Ceruzzi: *A history of modern computing*, S. 334.
[61] Raymond: *The Cathedral and the Bazaar*, S. 29.

3.4 Entstehung von Open-Source-Software

Parallel zu der Bewegung um Richard Stallman manifestierte sich die wirtschaftliche Unabhängigkeit der Software in den 1980er und 90er Jahren, während sich der Personal Computer weiter verbreitete, immer stärker. Nach dem Anbruch der 90er Jahre übertraf die Entwicklung und das Marketing von Software bald schon das der Hardware, die sich wiederum stärker zu einer billigen massenproduzierten Ware hin entwickelte:[62] „By the mid-1990s personal computers had become a commodity, allowing commercial software to come to the fore as the central place where innovation was conveyed to users."[63] Ausgehend von der Gemeinschaft der Freien Software führten unterschiedliche Interessen im Laufe der Zeit zu einer Spaltung in die Freie-Software- und Open-Source-Bewegung.

Die Geburtsstunde der Initiative rund um Open-Source-Software kann im sogenannten *Browserkrieg*, einer Auseinandersetzung zwischen den Unternehmen *Netscape* und *Microsoft*, gesehen werden. Vor der Einführung des *Internet Explorers* durch Microsoft besaß Netscape mit seinem Internet Browser *Netscape Navigator* einen Marktanteil von über 80%. Der ‚Krieg' zwischen den beiden Kontrahenten dauerte von 1995 bis 1998 an und endete darin, dass Netscape sich den Vorteilen, die der Internet Explorer seinen Usern bot, beugen und den Markt fast komplett seinem Konkurrenten Microsoft überlassen musste. Als Konsequenz daraus machte Netscape den Quellcode seines Browsers frei zugänglich und bewirkte damit wiederum das überaus erfolgreiche Open-Source-Projekt *Mozilla*.[64] Nur kurze Zeit später wurde die *Open Source Initiative* (OSI) von einer Gruppe von professionellen Softwareentwicklern gegründet. Als leitende Instanzen dieser Bewegung sind Eric Raymond und Bruce Perens zu nennen. Der Fall um Netscape hatte ihnen deutlich gemacht, dass sich mit Softwarenutzern und -entwicklern stärker auseinandergesetzt und eine engagierte Community aufgebaut werden muss, um den Quelltext einer Software erstellen und insbesondere verbessern zu können. Um diesen Ansatz weiterzuverfolgen, entschieden sie sich für ein Label, das sich von dem ethisch und politisch motivierten Begriff der Freien Software deutlich abgrenzt.[65]

---

[62] Vgl.: Korb: „Geschichte der Softwareprogrammierung".
[63] Ceruzzi: *A history of modern computing*, S. 345.
[64] Vgl.: Buxmann, Peter; Diefenbach, Heiner; Hess, Thomas: *Die Softwareindustrie. Ökonomische Prinzipien, Strategien, Perspektiven*. Berlin/Heidelberg: Springer 2011, S. 31f.
[65] Vgl.: Open Source Initiative: „History of the OSI", in: https://opensource.org/history (Abrufdatum: 05.01.2018).

Der leitende Gedanke der Open Source Initiative lässt sich an einer Stelle aus Raymonds Aufsatz *The Cathedral and the Bazaar* aufzeigen:

> Because source code is available, [users] can be *effective* hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your user will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.[66]

In demselben Aufsatz unterscheidet Raymond zwischen zwei möglichen Modellen der Softwareprogrammierung: einerseits *the cathedral*, das traditionelle Modell, das auf nur wenigen Programmierern beruht, die hierarchisch strukturiert arbeiten und eine Software erst zu einem späten Entwicklungsstand veröffentlichen; andererseits *the bazaar*, ein dezentrales Modell, das durch die Kooperation von einer Vielzahl von Programmierern entsteht, die die Software wiederum in den verschiedensten Stadien testen und bearbeiten.[67] Während *the cathedral* die kommerzielle Welt widerspiegelt, zeigt sich in *the bazaar* die Entstehungsweise von Torvalds Linux-Projekt, auf das Raymond sich in seinen Ausführungen immer wieder bezieht.[68] Raymond kommt zu dem Schluss, dass Bugs und Probleme bei der Entwicklung im Bazaar-Modell „flache Phänomene“[69] sind, die deutlich schneller gelöst und korrigiert werden, sobald sie vielen Mitentwicklern offengelegt werden. Gleichzeitig betont er allerdings, dass es kaum möglich ist, eine Software von Grund auf in diesem Modell zu entwickeln. Die Ausgangslage sollte immer ein schon existierender Quelltext sein, den man testen, korrigieren und verbessern kann.[70] Mit der Definition der Softwarekategorie *Open Source* legen Perens und Raymond auch einen Standard fest, der von einem Computerprogramm erfüllt werden soll. Softwarelizenzen müssen demnach an einer Reihe von Kriterien gemessen werden.[71]

---

[66] Raymond: *The Cathedral and the Bazaar*, S. 36.
[67] Vgl.: Ebd., S. 27ff.
[68] Tatsächlich offenbart er eine große Bewunderung für Torvalds: „In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model.“ Ebd., S. 37.
[69] Vgl.: Ebd., S. 41.
[70] Vgl.: Ebd., S. 57ff.
[71] Diese lauten wie folgt:
1. Freie Weitergabe: Die Lizenz darf niemanden daran hindern, die Software als Teil eines Paketes, das Programme aus unterschiedlichen Quellen enthält, zu verschenken oder zu verkaufen. **2.** Quellcode: Das Programm muss den Quellcode enthalten und eine Weitergabe dessen erlauben. **3.** Abgeleitete Werke: Die Lizenz muss Änderungen und Ableitungen der Software zulassen und diese Software unter den gleichen Bedingungen wie die Lizenz der ursprünglichen Software verteilen lassen. **4.** Intaktheit des Quellcodes des Autors: Die Lizenz muss die Verteilung von Software, die aus verändertem Quellcode gebaut ist, explizit erlauben. **5.** Keine Diskriminierung von Personen oder Gruppen: Die Lizenz darf keine Person oder Personengruppe diskriminieren. **6.** Keine Diskriminierung von Einsatzfeldern: Die Lizenz darf niemanden daran hindern, das Programm in einem bestimmten Feld zu gebrauchen. **7.** Weitergabe der Lizenz: Die an das Programm gebundenen Rechte müssen für alle Personen gelten, die das Programm erhalten, ohne dass für diese eine zusätzliche Lizenz erforderlich ist. **8.** Die Lizenz darf nicht auf ein Produkt beschränkt sein: Die an das Programm gebundenen Rechte dürfen nicht davon abhängen, dass das Programm Teil einer bestimmten Softwareverteilung ist. **9.** Die Lizenz darf keine andere Software einschränken: Die Lizenz darf

Im Vergleich zur Freien Software stellt die Open Source Bewegung den Aspekt der kommerziellen Nutzung sowie des kollaborativen Entwickelns noch stärker heraus. Einige GNU-Programme etwa wurden von Stallman alleine entwickelt und erst in einer fortgeschrittenen Version herausgegeben. Was die Kategorisierung als Freie Software rechtfertigt, ist alleine die Veröffentlichung des Programms samt Quellcode in einer GPL oder ähnlichen Lizenz. Für Torvalds hingegen war es in den frühen Jahren üblich, mehrmals täglich einen neuen, unfertigen Kernel zu veröffentlichen. Dadurch, dass er ein Fundament an Mitentwicklern pflegte und das Internet für seine Kollaborationen wirksam einsetzte, konnte er mit dieser Methode Erfolg erzielen. Mit dem Motto „Release Early, Release Often" orientiert sich die Open Source Initiative stark an Torvalds Vorgehen.[72] Diese Bestrebungen gingen jedoch nicht ohne Widerstand von der anderen, konventionellen Seite einher. Im Sommer 1998 gab Microsoft intern eine Studie in Auftrag, die das Phänomen Linux sowie mögliche Strategien gegen den neuen Konkurrenten untersuchen sollte. Kurz darauf geriet das sogenannte *Halloween Document* an die Öffentlichkeit und gab darüber Auskunft, dass Microsoft beabsichtige, künstliche Produktinkompatibilitäten zu entwickeln, die es seinen Kunden erschweren sollten, Linux als Alternative zu *Windows* auf Rechnern installieren oder benutzen zu können.[73] Bereits zuvor hatte Microsoft öffentlich die Meinung kundgetan, Software mit offenliegendem Quelltext würde die Leistungen seiner Entwickler nicht ausreichend honorieren.[74] Auch wenn mit dem Durchsickern des Halloween Documents deutlich wurde, dass der Softwaremonopolist Microsoft dazu bereit war, seinen Marktstatus in jedem Fall aufrechtzuerhalten, änderte sich am Erfolg der Open-Source-Bewegung allerdings nichts, wie Raymond dokumentiert:

> In 1998 and late 1999, we've seen a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.[75]

---

keine Einschränkungen gegenüber anderer Software machen, die zusammen mit der lizenzierten Software weitergegeben wird. **10.** Die Lizenz muss neutral gegenüber Technologie sein: Keine Verfügung der Lizenz darf auf einer individuellen Technologie oder Art des Interfaces basieren.
Vgl. hierzu: Open Source Initiative: „The Open Source Definition", in: https://opensource.org/osd (Abrufdatum: 05.01.2018).

[72] Vgl.: Raymond: *The Cathedral and the Bazaar*, S. 37ff.

[73] Vgl.: Pfaffenberger, Bryan: „The Rhetoric of Dread. Fear, Uncertainty, and Doubt (FUD) in Information Technology Marketing", in: *Knowledge, Technology & Policy*, Band 13, Heft 3 (2000), S. 78-92, hier S. 88f.

[74] Vgl.: Ceruzzi: *A history of modern computing*, S. 337.

[75] Raymond: *The Cathedral and the Bazaar*, S. 78.

4 Machtverhältnisse I – Programmierung von Software

Im folgenden Teil soll sich der Blickpunkt auf Freie Software ändern. Bisher wurde diese innerhalb eines Rahmens diskutiert, der historisch, ökonomisch und rechtlich bestimmt und deshalb nicht anfechtbar ist. Es ist sozusagen eine Bestandsaufnahme erfolgt, die dazu gedient hat, das Umfeld, in dem die Freie Software entstanden ist und in dem sie sich in Teilen heute noch bewegt, darzulegen. Blickt man über diese Aspekte hinaus, lässt sich aber auch von einem anderen Standpunkt aus über Freie Software und Software im Allgemeinen sprechen. Dieser Teil der Arbeit macht es sich zur Aufgabe, die mögliche Freiheit des Computerprogrammes unter medienwissenschaftlichen, techniknäheren Faktoren zu untersuchen. Schnell kommt in einem solchen Diskurs die Frage auf: Wie frei kann Software eigentlich sein, wenn man sie in ihrem Verhältnis zu dem Computer[1], von dem sie ausgeführt wird, begreift? Mit verstärktem Blick auf die Hardwareseite soll den Spuren dieser Frage im Folgenden nachgegangen werden – zuerst auf der Ebene der Softwareprogrammierung. Hier bieten sich grundlegend zwei Faktoren dar, die eine Betrachtung verlangen: der Gebrauch einer Programmiersprache und der Prozess des Kompilierens. Es soll dabei aufgezeigt werden, dass schon im Akt des Programmierens eine maschinelle Macht über Handlungsmöglichkeiten zum Ausdruck kommt.

4.1 Programmiersprache

Der Softwareentwickler Erik Möller schrieb einmal: „Computer sind stumpfe Rechenknechte, die alles tun, was man ihnen sagt, wenn man nur ihre Sprache spricht.“[2] Daran werden zwei Dinge zugleich deutlich. Erstens: Es ist möglich, den Computer zu jedem berechenbaren Zweck zu instruieren. Dies ist die Idee, die mit der Turingmaschine geboren und mit einigen der frühesten Digitalcomputer erstmals umgesetzt wird: Der universellen Rechenmaschine ist es möglich, jede denkbare Rechnung durchzuführen, alles, was berechenbar ist, zu berechnen. In einem weiteren Sinn bedeutet das also, dass der Nutzer den Rechner das ausführen lassen kann, was auch immer ihm beliebt. Allerdings konstatiert Möller zweitens, dass dieser Nutzer, um überhaupt Herr über die in der Maschine ablaufenden Rechenvorgänge werden zu können, zunächst einmal dessen

[1] Der Computer-Begriff wird in der folgenden Ausführung generisch verwendet und meint im Wesentlichen den Personal Computer.
[2] Möller, Erik: *Die heimliche Medienrevolution. Wie Weblogs, Wikis und freie Software die Welt verändern*. Hannover: Heise 2006, S. 56.

‚Sprache' lernen muss. Um den Computer zu instruieren, d.h. programmieren zu können, wird also die Nutzung eines spezifischen Codes vorausgesetzt. Dieser wiederum wird für gewöhnlich mithilfe von Programmiersprachen formuliert. Welche Programmiersprache dabei im Detail verwendet wird, spielt zunächst keine Rolle, solange eine Maschine nur Turing-vollständig ist; die Menge an verschiedenen Programmiersprachen ist in der heutigen Zeit unüberschaubar groß. Was sie alle eint, ist die formale Syntax, durch die sie geregelt sind. Denn: „[…] in order to be useful in computing, a language needs to be abstract, or formalized according to a mathematical model."[3] Um Rechenvorschriften formulieren zu können, die wiederum vom Computer ausgeführt werden können, braucht es also ein vorgegebenes Muster. Stellt man dem im Vergleich eine natürliche Sprache gegenüber, so fällt auf, dass diese oft nicht eindeutig klar ausgelegt werden kann. Viele Äußerungen sind sowohl in ihrer Semantik wie auch Syntax mehrdeutig und lassen daher einen Raum für Interpretationen. Für die Beschreibung von Computeranweisungen sind natürliche Sprachen daher kaum dienlich. Formale Sprachen hingegen ermöglichen es bei der Programmierung, deterministisch nach Regeln vorzugehen, indem „[…] auf der Basis einer relativ kleinen Menge von syntaktischen Regeln Programme formuliert werden, die eine Semantik aufweisen, die von Rechnern weitgehend verstanden wird."[4]

Die Kategorisierung von formalen Sprachen geht zurück auf Noam Chomsky, welcher 1956 auf der Basis von Ersetzungs- und Transformationsregeln ein Modell – ursprünglich sollte es der Beschreibung von natürlichen Sprachen dienen – zur Untersuchung formaler Grammatiken entwarf.[5] Insgesamt klassifizierte er vier Grammatiktypen, die Komplexitätsklassen bilden: Je nach der Höhe der Einschränkungen für ihre Produktionsregeln reichen sie von *Typ 0* – unbeschränkte Grammatik – bis zu *Typ 3* – am stärksten beschränkte Grammatik.[6] Sein Konzept dient als fundamentale Basis dessen, was eine formale Sprache darstellt: Sie ist mathematisch beschreibbar, indem die Semantik formalisiert und eine bestimmte Menge von Zeichenketten aus einem Zeichenvorrat miteinander kombiniert wird. So ist ein *Alphabet V* hier klar als endliche, nichtleere Menge von Zeichen definiert. Im Rahmen eines Alphabets werden einzelne Zeichen miteinander verkettet, um neue Zeichenketten zu generieren. Zieht man als

[3] Frabetti, Federica: *Software Theory. A Cultural and Philosophical Study*. London/New York: Rowman & Littlefield International 2015, S. 133.
[4] Hedtstück, Ulrich: *Einführung in die Theoretische Informatik. Formale Sprachen und Automatentheorie*. Berlin/Boston: De Gruyter 2012, S. 5.
[5] Vgl.: Frabetti: *Software Theory*, S. 44.
[6] Vgl.: Hedtstück: *Einführung in die Thoretische Informatik*, S. 32f.

Beispiel das Alphabet $V = \{0,1\}$ heran, so folgt daraus:

> Die Wörter über V sind alle endlichen 0-1-Folgen inklusive das Leerwort ε, d.h. $V^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots\}$. Eine typische Sprache L über V ist die Menge aller vorzeichenlosen Binärzahlen ohne führende Nullen: $L = \{0, 1, 10, 11, 100, 101, 111, 1000, 1001, \ldots\}$.[7]

Weil sie sich zur mathematisch präzisen Beschreibung und Spezifizierung eignet, ist die Bedeutung einer Programmiersprache also absolut klar und ruft ein eindeutiges Maschinenverhalten hervor – vorausgesetzt, sie wird richtig verwendet. Raum für Interpretationen gibt es hier nicht; entweder der Code funktioniert oder er funktioniert nicht.[8]

Es ist eine Tatsache, dass die Möglichkeit der Programmierung durch die Verwendung einer Programmiersprache bedingt wird, deren symbolische Ordnung von der Architektur des Rechners vorgegeben wird. Sie kann von einem Programmierer nur adaptiert werden, um der Maschine Befehle erteilen zu können. Das Kommunikationsverhältnis wird also vom Rechner her bestimmt, wie auch Markus Krajewski festgestellt hat: „Letztlich gehorchen sie [Operateure, Programmierer, Benutzer] jedoch […] allesamt den Regeln, welche die Maschine vorgibt, obwohl sie selbst diese nominell mit Kommandos steuern."[9] Um die Zügel der Operationen überhaupt erst in der Hand halten zu können, muss sich der Programmierer also zunächst einmal der bestehenden, an dieser Stelle sprachlichen Hierarchie hingeben.

4.2 Kompilierung

An dieser Stelle soll noch einmal kurz das bereits genannte Zitat Möllers von Computern als „[…] stumpfe Rechenknechte, die alles tun, was man ihnen sagt, wenn man nur ihre Sprache spricht."[10] in Erinnerung gerufen werden. Es erscheint nützlich, das Sprechen der ‚Computersprache' an dieser Stelle weiter auszudifferenzieren. Grundlegend lässt sich zwischen Quelltext in einer höheren Programmiersprache, Quelltext in einer

[7] Hedtstück: *Einführung in die Theoretische Informatik*, S. 7.
[8] Vgl.: Ebd., S. 5ff.
[9] Krajewski, Markus: „Dienstleistungsagenturen. Zur Delegation von Handlungsmacht zwischen Subalternen und Software-Services", in: Mersch, Dieter; Paech, Joachim (Hrsg.): *Programm(e)*. Zürich/Berlin: diaphanes 2004, S. 125-157, hier S. 147.
[10] Möller, Erik: *Die heimliche Medienrevolution*, S. 56.

Assemblersprache sowie einem binären Maschinencode unterscheiden.[11] Bei dem für die meisten modernen Programme verwendeten Programmcode handelt es sich um einen Quelltext, der in einer höheren Programmiersprache geschrieben wurde. Dieser ist in eben dieser Form für die Maschine aber keineswegs les- und ausführbar, denn sie verlangt wiederum nach einem Maschinencode, den sie in seiner binären Form auslesen und demnach schließlich in elektronische Zustände umwandeln kann. Dieser Maschinencode, der vom Prozessor verarbeitet wird und letztlich zur Steuerung der Maschine führt, ist für den Menschen prinzipiell zwar verständlich, eine Programmierung in Binärcode ist aufgrund der Komplexität von modernen Rechnern heutzutage jedoch mit einem sehr hohen Aufwand verbunden. Es lässt sich festhalten, dass zwischen Quell- und Maschinencode eine große Distanz herrscht und es einer Vermittlung zwischen den beiden Seiten, sozusagen einer Übersetzung in den jeweils ‚nativen' Code bedarf. Hier kommt der Compiler ins Spiel: Er tritt als vermittelnde Instanz zwischen den Quellcode (Programmierer) und Maschinencode (Computer) und leistet eine semantische Übersetzung der sprachlichen Logik des Programmierers in die Maschinensprache des jeweiligen Computer- bzw. Prozessortyps: „Simply stated, a compiler is a program that can read a program in one language – the source language – and translate it into an equivalent program in another language – the target language […]."[12] Erst durch diese kompilierende Instanz wird die Implementierung einer Sprache möglich, die maschinell lesbar ist.[13]

Was wie ein einfacher, einmaliger Übersetzungsprozess klingen mag, ist in der Tat ein äußert vielschichtiger Vorgang. Zu Beginn des Kompilierens steht die Lexikalische Analyse, auch *scanning* genannt. Dabei werden in der Zeichenkette Unterbrechungen in Form von Leerzeichen oder Semikola gesucht und erkannt, sodass die einzelnen Symbole in kleinere Bedeutungseinheiten, auch *Lexeme* genannt, gruppiert werden können. Bei der Syntaxanalyse, auch *parsing* genannt, erfolgt dann eine Umordnung der Lexeme nach den syntaktischen Regeln der jeweiligen Grammatik. Nach einer semantischen Analyse

[11] Die ‚Höhe' einer Programmiersprache lässt sich auf die Höhe ihres Abstraktionsgrades von der Maschinenebene zurückführen: „Je höher und komfortabler die Hochsprachen, desto unüberbrückbarer ihr Abstand zu einer Hardware […]." Kittler, Friedrich A.: „Protected Mode", in: Ders.: *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993, S. 208-224, hier S. 210.

[12] Aho, Alfred V. [u.a.]: *Compilers. Principles, Techniques, & Tools*. Boston [u.a.]: Pearson Addison-Wesley 2007, S. 1.

[13] Das Kompilieren ließe sich im weitesten Sinne auch als Hermeneutik im Computer verstehen. Denn durch seine Funktion als Übersetzer ermöglicht es der Compiler, dass der vom Programmierer geschriebene Quelltext eine Auslegung durch die Maschine erfährt.

wird nun zunächst ein Zwischencode, der *intermediate code*, erzeugt. Es erfolgt eine Code-Optimierung und schließlich die finale Code-Erzeugung, indem Register und Speichersätze für jede vom Programm verwendete Variable ausgewählt und die Befehle aus der zeitweiligen Repräsentation des Ursprungscodes in Folgen von Instruktionen und damit in den als Zielcode bestimmten Maschinencode übersetzt werden.[14] Verfolgt man diesen Prozess, so lassen sich hierbei nicht bloß einen, sondern gleich mehrere Brüche im Code feststellen. Denn von seiner anfänglichen Form als Quellcode bis hin zum Maschinencode erfährt dieser eine ständige Zersetzung. Die italienische Medienwissenschaftlerin Federica Frabetti bezeichnet dies als einen Prozess der permanenten Umschreibung bzw. Wiedereinschreibung: „A compiler is thus […] an automaton that ‚reads' any program previously written in the high-level programming language for which the compiler has been designed (the ‚source language') and reinscribes it – usually – as binary code (the ‚target language')."[15] Ebenso betont sie die Vielschichtigkeit des Vorgangs: „In order to be reinscribed *as* circuits […], this program needs to undergo not just one but many reinscriptions."[16] Dem Zwischencode misst Frabetti eine Rolle zu, die zentral ist für eine Exteriorisierung der Software:

> The intermediate code is a further reinscription of the source code, a further ‚exteriorization' of the software system, and a further transformation of the rupture inscribed in the source program – ruptures that actually make it possible for code to function.[17]

Sie spricht hier einen Bruch an, der im Quellcode gegeben ist. Gemeint sind Unterbrechungen, die, wie bereits erwähnt, in Form eines Leerzeichens oder Semikolons etwa die Kette von Symbolen im Text gliedern. Mit dem Kompilieren werden auch diese Unterbrechungen fortgeschrieben, welche dem Umgruppieren der Symbole in neue Lexeme dienen. Diese Brüche im Code sind demnach notwendig für das Funktionieren des Programms, da die Instruktionen ohne diese Unterbrechungen nicht lesbar und ausführbar sind. Zusätzlich kann aber auch ein weiterer Bruch im Programmcode ausgemacht werden, welcher durch die Übersetzungsprozesse stets neu zersetzt wird. Denn festzuhalten bleibt, dass der von Frabetti dargelegte Prozess der permanenten Umschreibung in einer materiellen Transformation des Programmcodes besteht.[18]

---

[14] Vgl.: Fischer, Peter; Hofer, Peter: *Lexikon der Informatik*. Berlin/Heidelberg: Springer 2011, S. 181. Sowie: Aho [u.a]: *Compilers*, S. 5ff.
[15] Frabetti: *Software Theory*, S. 145.
[16] Ebd., S. 144.
[17] Ebd., S. 146.
[18] Vgl.: Ebd., S. 145f.

Mit Blick auf die Übersetzung als materielle Transformation des Codes lässt sich demnach behaupten, dass das, was zu Beginn von einem Programmierer geschrieben wird, lediglich als Programm*entwurf* gelten kann,[19] welcher im weiteren Prozess, also im Durchlaufen des Compilers, eine grundlegende Übersetzung in einen ausführbaren Maschinencode erfährt. Die Bedeutung der konstanten Ein- bzw. Umschreibung nach Frabetti wird hieran augenscheinlich: aus einem Entwurf – je höher die Programmiersprache, desto höher auch der Abstraktionsgrad von der Maschinenebene – werden erst nach Durchlaufen des Compilers operative Rechenanweisungen und letztlich ein Programm, das der Prozessor verarbeiten kann. Wolfgang Hagen sieht in den Übersetzungen, die der Compiler leistet, „eine buchstäblich ideelle und trügerische Kontinuität“[20]:

> Zwischen einem abstrakten Computer, den jedes kunst- oder ‚hoch'-sprachliche Programm repräsentiert, und einer realen Maschine, die gesteuert werden soll, steht, wem sage ich das, wenigstens ein Generationsprozeß einer weiteren Maschine. Einige deutsche Informatiklehrbücher nennen diese generische Maschine, die aus dem Quellcode den Maschinencode kompiliert, den ‚Übersetzer'. Was die Sprache der abstrakten Computer betrifft, also die ‚Sourcen', so wirkt dieser Übersetzer wie ein Übersetzen über jenen unterirdischen Fluß, den die Griechen mit dem Reich des Todes identifizieren. Der Übergang von einem symbolischen Programmtext zum realen Maschinencode eines Programms tötet die Sprache, die es in Gang setzt, ab.[21]

Der Übersetzungsweg von der einen in die andere Sprache stellt also keineswegs einen gleichmäßigen Fortgang dar. Denn schließlich entsteht – mit Hagens Worten – „eine andere, neue Sprache, die mit der Ursprungssource nur noch wenig bis nichts zu tun hat“[22].

Hagen spricht in dem obenstehenden Textauszug zudem ein Paradox an, das sich in Bezug auf die Softwareprogrammierung feststellen lässt: Um Software programmieren zu können, ist eine Software nötig. Es lässt sich gewissermaßen behaupten, dass das Computerprogramm überhaupt erst die Vorbedingung für die Schaffung eines neuen Programms bildet. Dies wird am Compiler deutlich, ohne welchen es nicht möglich ist,

---

[19] Frabetti schreibt hierzu: „What has been written is generally a large ensemble of interconnected programs, expressed in some high-level programming language.“ Frabetti: *Software Theory*, S. 143.

[20] Hagen, Wolfgang: „Der Stil der Sourcen. Anmerkungen zur Theorie und Geschichte der Programmiersprachen“, in: Warnke, Martin; Coy, Wolfgang; Tholen, Georg C. (Hrsg.): *HyperKult. Geschichte, Theorie und Kontext digitaler Medien*. Basel/Frankfurt am Main: Stroemfeld 1997, S. 33-68, hier S. 35.

[21] Ebd.

[22] Ebd.

einen Programmentwurf in Maschinenbefehle zu übersetzen. Auch Frabetti hat diesen Umstand thematisiert: „[…] it is worth noting how, being computer programmes, compilers are themselves written in programming languages, which in turn have been specified and (typically) compiled. Thus, in principle it takes a compiler to generate a compiler.“[23] Und noch mehr als dies: Bevor ein Compiler der Programmierung vorausgesetzt werden kann, ist ein Texteditor erforderlich, ohne welchen ein Textentwurf erst gar nicht aufgeschrieben werden kann. Erst im weiteren Schritt ist dann die Übersetzung durch den Compiler möglich. Um die Lücke zwischen Mensch und Maschine in differenzierten Ebenen zu überbrücken, sind diese Programme eine dringende Voraussetzung, der sich bei der Programmierung nur schwer entzogen werden kann. Durch die Komplexität der modernen Computerarchitektur wird demnach eine Abhängigkeit von Programmen wie dem Compiler erzeugt, die den Programmierer folglich von der Hardware entfernen. Mit den Worten Kittlers lässt sich dies weiter ausführen:

> Es war die schiere Unmöglichkeit, über alle Schaltzustände einer Computerhardware Bescheid zu wissen, die schon in der Frühzeit ihrer Technologie zur Einführung von Abstraktionen geführt hat. Das menschliche Kurzzeitgedächtnis ist in dramatischem Unterschied zu Turingmaschinen außerstande, kombinatorische Explosionen zu speichern. Also stellten höhere Programmiersprachen, deren Konstrukte von einem maschinennäheren Programm, dem Interpreter oder Compiler, erst noch in ausführbaren Code übersetzt werden müssen, die Grundlage her, auf der sich mittlerweile ganze Türme von Softwarehierarchien errichtet haben.[24]

Ohne Compiler u. ä.[25] ist es dem Programmierer also kaum möglich, eine Zugänglichkeit zu erreichen, die die Voraussetzung dafür schafft, den Computer instruieren zu können. Er ist demnach auf die Metaphorik des Computers und eine Abstraktion der Prozesshaftigkeit angewiesen, denn sonst kann er die in der Maschine ablaufenden Prozesse nicht verstehen.[26] Insbesondere die Verwendung von höheren Programmiersprachen entfernt ihn mehr von der Hardware und lässt diese in gewisser Hinsicht in Vergessenheit geraten. Ein direkter Zugriff wird verwehrt, mit dem Ergebnis, dass die Software den Menschen von der Hardware trennt. Dies wird – um ein Beispiel

---

[23] Frabetti: *Software Theory*, S. 137.
Sie verweist hier außerdem auf den 1970 entwickelten Compiler *Yacc*, dessen Name eine Einschreibung dieses Umstandes darstellt: Yacc steht für „Yet Another Compiler Compiler“.

[24] Kittler, Friedrich A.: „Hardware, das unbekannte Wesen“, in: Krämer, Sybille (Hrsg.): *Medien, Computer, Realität. Wirklichkeitsvorstellungen und Neue Medien*. Frankfurt am Main: Suhrkamp 1998, S. 119-132, hier S. 125.

[25] Zu nennen wäre hier außerdem der *Interpreter*, welcher sich im Vergleich zum Compiler durch eine andere Funktionsweise auszeichnet, dabei aber das gleiche Ergebnis hervorbringt.

[26] Vgl.: Krajewski: „Dienstleistungsagenturen“, S. 138f.

aus der Praxis zu nennen – schon daran deutlich, dass das, was heute unter Entwicklern umgangssprachlich als Programmieren in Maschinencode bezeichnet wird, tatsächlich Programmieren in Assemblersprache meint, welche für den Menschen weitaus einfacher zu verstehen ist. Denn es ist eine Tatsache, dass kaum ein Programmierer seinen Computer heutzutage mehr per Maschinencode instruieren kann. Er kann lediglich Einfluss auf die Gestaltung des Quellcodes nehmen, was der Blick auf das Kompilat verdeutlicht: Das Compiler-Konstrukt gestaltet sich – bei absolut gleichbleibendem Quelltext eines ausgewählten Programms – nach dem Durchlaufen des Compilers grundlegend anders, wenn das Betriebssystem oder der Compiler ausgetauscht werden. Auch wenn sich der Inhalt des Programms – in Bezug auf das, was das Programm letztlich hervorbringt – hierbei nicht verändert, so ändert sich seine Gestalt doch radikal. Dabei wird der Programmierer gewissermaßen degradiert, denn schlussendlich bringt der Compiler als sein Ergebnis Maschinenbefehle hervor, die in ihrer Form nicht beeinflussbar sind. Der Handlungsspielraum in der Programmierung begrenzt sich demnach auf die Sphäre von Quell- und gegebenenfalls Assemblercode; wie sich der Binärcode aber im Detail zusammengesetzt, wird vom Compiler festgelegt. Auf das Kompilat kann der Programmierer schlussendlich nur indirekt Kontrolle ausüben.[27]

Um diesen Punkt weiter zu verdeutlichen, kann abschließend gefragt werden, wann und wo genau der Akt der Programmierung stattfindet, wenn ein Compiler im Gebrauch ist. Wird schon dann programmiert, wenn der Entwurf eines Programmcodes in einem Editor aufgeschrieben wird? Oder viel eher dann, wenn dieser in eine ausführbare Datei kompiliert wird? Oder letzten Endes nicht dann, wenn aus dem Binärcode in der Maschine Zustände werden, die in einer Aktivität münden? Auch wenn diese Problematik an dieser Stelle nur kurz angeführt werden soll und eine detailliertere Betrachtung erforderlich wäre, um eine fundierte Einschätzung zu geben, wird bereits oberflächlich deutlich, dass es kaum möglich ist, diese Frage mit absoluter Sicherheit beantworten zu können. Und gerade die Unmöglichkeit, zu wissen, wo genau die Programmierung stattfindet, verdeutlicht wiederum die Kluft, die sich zwischen Mensch und Maschine auftut. „Wir können schlichtweg nicht mehr wissen, was unser Schreiben tut, und beim Programmieren am allerwenigsten.“[28], schreibt auch Kittler. Einerseits lässt sich also

[27] Er benötigt hierfür zum Beispiel einen Hex-Editor, welcher ihn dazu befähigt, Binärdateien zu analysieren, aber auch zu manipulieren.
[28] Kittler, Friedrich A.: „Es gibt keine Software“, in: Ders.: *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993, S. 225-242, hier S. 229.

schließen, dass die Software den Programmierer von der Hardware trennt, indem sie sich zwischen diese beiden Komponenten stellt und als ‚vermittelnde' Instanz die direkte Handlungsfreiheit eingrenzt. Im Umkehrschluss bedeutet dies aber auch, dass stets eine gewisse Vorrangigkeit der Hardware gegenüber der Software besteht.

5 Machtverhältnisse II – Ausführung von Software

Nach dem Prozess des Kompilierens erfolgt die Ausführung des Programms durch den Computer. Zentral hierfür ist der Prozessor, welcher gewissermaßen den Kern der Maschine bildet. Denn er beinhaltet das Rechenwerk, d.h. das Schaltwerk zur Ausführung der Maschinenbefehle, sowie das Steuerwerk, auch Leitwerk genannt, d.h. Befehlszähler, -register und -decoder. Während das Steuerwerk aus dem Schaltwerk besteht, welches Signale überwacht und koordiniert sowie den Kontrollfluss beeinflusst, stellt das Rechenwerk eine Verarbeitungseinheit dar, die Algorithmen verrechnet. Gleichbedeutend mit dem Prozessor wird häufig auch der Begriff *CPU*, kurz für *central processing unit* oder zentrale Verarbeitungseinheit, verwendet:[1] „Die CPU […] ist das ‚Gehirn' des Computers. Sie führt die im Hauptspeicher abgelegten Programme aus, indem sie deren Befehle nacheinander abruft, analysiert und dann ausführt."[2], so Tanenbaum.

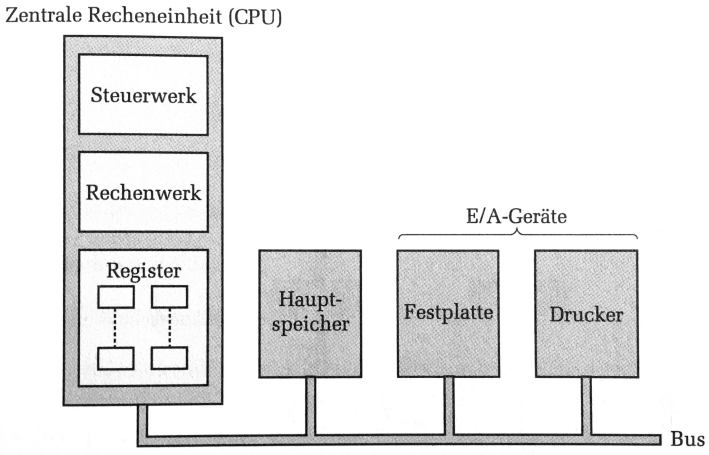

Abb. 5.1: Der Aufbau der CPU im Überblick. Hinzu kommen beim einfachen Computer ein Hauptspeicher sowie die Ein- und Ausgabegeräte.

Mit Blick auf die vorangegangenen Schritte lässt sich verzeichnen, dass die Abfolge von Befehlen, die algorithmisch im Computerprogramm festgehalten wurden, an dieser Stelle – sowohl zeitlich als auch örtlich – von der Software- auf die Hauptwarekomponente übergehen. Durch die Darlegung der technischen Begebenheiten liegt schon auf der Hand, dass sich rein materiell eine Vorherrschaft durch die Hardwarekomponenten

[1] Vgl.: Fischer; Hofer: *Lexikon der Informatik*, S. 710, S. 737 und S. 864. Sowie: Tanenbaum, Andrew S.: *Computerarchitektur. Strukturen – Konzepte – Grundlagen*. München: Pearson 2006, S. 71f.
[2] Tanenbaum: *Computerarchitektur*, S. 71.

ergibt. Der Rahmen für die Ausführung des Programms wird durch sie abgesteckt und damit im weiteren Sinne auch die Grenzen für einen Handlungsspielraum des Users festgelegt. Auch durch Kittlers Werke ist bekannt, dass „[g]ute Gründe […] für die Unabdingbarkeit und folglich auch die Vorgängigkeit von Hardware [sprechen]“[3]:

> Das AirLand Battle 1991 hat es einmal mehr gezeigt: Unter den postmodernen Strategien des Scheins ist keine so wirksam wie die Simulation, daß es Software überhaupt gibt. Bis zum Gefechtsfeldbeweis des Gegenteils, als Computer unzweideutig zu erkennen gaben, Hardware zur Zerstörung irakischer Hardware […] zu sein, verbreiten Werbeprospekte und Medienkonferenzen das Märchen von einer Softwareentwicklung, die schon immer sanfter und benutzerfreundlicher, spiritueller und intelligenter geworden wäre, bis sie eines Tages den Deutschen Idealismus effektiv heraufführen, also Mensch werden würde. Weshalb Software, dieses Milliarden-Dollar-Geschäft mit einem der billigsten Elemente dieser Erde, nichts unversucht läßt, um besagte Menschen an die entsprechende Hardware gar nicht erst heranzulassen. Man kann es mit WORD 5.0 auf einem No Name AT 386 und (wie es so schön heißt) unter Microsoft DOS 3.3 über eben diese drei Wesenheiten ganze Aufsätze schreiben, ohne die Strategie des Scheins auch nur zu ahnen. Denn man schreibt – das ‚Unter‘ sagt es schon – als Subjekt oder Untertanen der Microsoft Corporation.[4]

Die folgende Ausführung soll die Vorrangigkeit der Hardware weiter verdeutlichen.

5.1 *Protected Mode* und Multitasking-Betrieb

Der im Zentrum von Kittlers Kritik stehende *Protected Mode* hat seinen Ursprung im Jahr 1982, als Intel den 80286-Prozessor, kurz auch 286 genannt, auf den Markt bringt. Mit 16 Bit und 134.000 Transistoren handelt es sich zu der Zeit um einen leistungsstarken Prozessor, der in zahlreichen PCs verbaut wird.[5] Er ist der erste Mikroprozessor, der über den sogenannten *Protected Mode* verfügt – einen zusätzlichen, abgesicherten Betriebsmodus, der es verhindert, dass betriebsnotwendiger Speicher überschrieben wird und der dadurch die Stabilität des Betriebssystems zu jeder Zeit sicherstellt. Mehr noch als in der Speicherkapazität liegt im Protected Mode der grundlegende Unterschied des 286-Prozessors zu seinen Vorgängern.[6] Vorangegangene Modelle, etwa der 8086, laufen

[3] Kittler: „Es gibt keine Software“, S. 237.
[4] Kittler: „Protected Mode“, S. 208.
[5] Vgl.: Intel Corporation: „Intel Timeline. A History of Innovation“, in: https://www.intel.com/content/www/us/en/history/historic-timeline.html (Abrufdatum: 08.12.2017).
[6] „Der Virtuelle Adressmodus des 80286 mit seinem Schutzkonzept und seiner Multitaskingfähigkeit erweitert die Einsatzmöglichkeiten eines Mikrocomputers beträchtlich und ist das eigentlich Neue am Mikroprozessor 80286.“ Schief, Rudolf: *Einführung in die Mikroprozessoren und Mikrocomputer. Am Beispiel der Mikroprozessoren 8080, 8085, Z80, 8086/8088, 80286, 80386*. Tübingen: Attempto 1991, S. 324.

demnach stets im *Real Mode*, wobei diese Unterscheidung erst mit der Einführung des 286-Prozessors getroffen werden kann.[7] An den erweiterten Namen *Real Address Mode* und *Protected Virtual Mode* werden die Unterschiede zwischen den Betriebsmodi deutlich: Befindet sich der Prozessor eines Mikrocomputers im Real Mode, kann auf den physikalisch-realen Speicher, d.h. auf den Arbeitsspeicher oder *Random-Access Memory* (RAM) zugegriffen werden. Rein physikalische Adressen können während der Nutzung demnach manipuliert werden. Das bedeutet, dass der virtuelle Adressraum identisch mit dem physikalischen Adressraum ist. Im Protected Mode hingegen haben Programme keinen Zugriff auf den physikalischen Adressraum, denn die Speicheradressierung erfolgt in Form von virtuellen Adressen. Durch Übersetzungsmechanismen auf den physischen Adressraum entsteht ein viel größerer virtueller Speicherraum.[8] Die Verwaltung des virtuellen Speichers sowie die Zuordnung der virtuellen zu den realen Adressen übernimmt die sogenannte *Memory Management Unit* (MMU), die auf dem Chip des Prozessors integriert ist.[9]

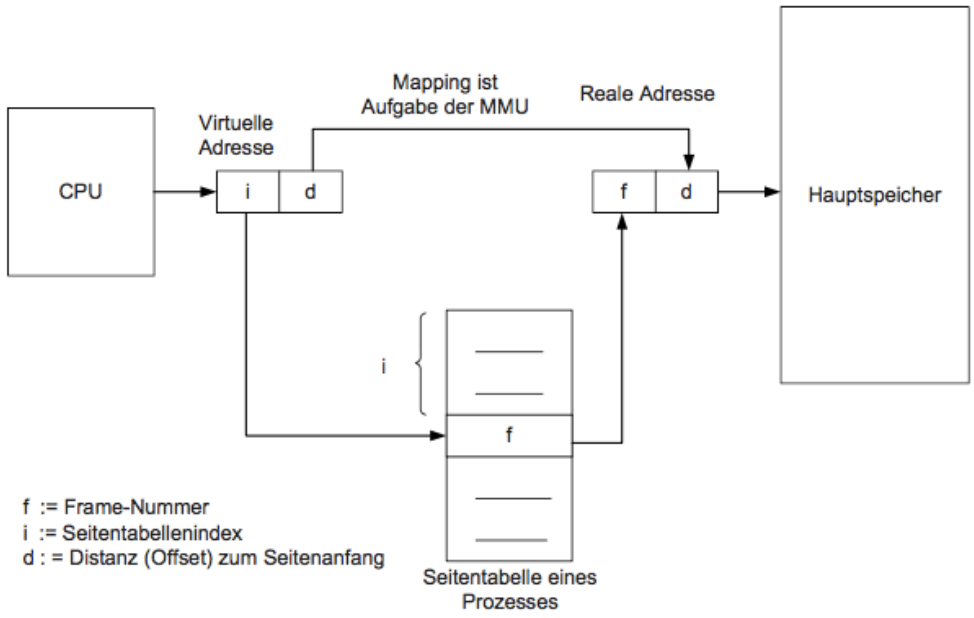

Abb. 5.2: Bei der virtuellen Speicherverwaltung erfolgt eine Zerlegung der virtuellen Adresse in eine Seitentabelle und in eine Distanz. Die Seitentabelle enthält Informationen über den tatsächlichen Ort der Frames, also Seiten-Plätze, im Hauptspeicher; die Distanz steht für die genaue Byteadresse innerhalb der Seite. Hinzu kommt ein Seitentabellenindex, der dabei hilft, in der

[7] In dem von Intel herausgegebenen Programmierhandbuch heißt es dann aber konkret: „The 80286 can be operated in either of two different modes: Real Address Mode or Protected Virtual Mode (also referred to as Protected Mode).“ Intel Corporation: „80286 and 80287 Programmer's Reference Manual“, 1987, in: http://bitsavers.trailing-edge.com/components/intel/80286/210498-005_80286_and_80287_Programmers_Reference_Manual_1987.pdf (Abrufdatum: 16.03.2018), S. 1-2.
[8] Vgl.: Intel Corporation: „80286 and 80287 Programmer's Reference Manual“, S. 2-1 und S. 2-23.
[9] Vgl.: Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 325ff.

Seitentabelle den Verweis auf die Frame-Nummer ausfindig zu machen. Die Verbindung von Distanz und Frame-Nummer ergeben schließlich die physikalische Adresse.

Der Unterschied liegt also ausschließlich in der Adressierung; der Protected Mode weist geschützte Adressräume auf, sodass letztendlich die CPU die Kontrolle über die Speicherzugriffe innehat. Prinzipiell könnte ein Real Mode Programm auch im Protected Mode laufen, da die CPU in beiden Fällen mit den gleichen Befehlen arbeitet.[10] Im Real Mode beläuft sich der Registersatz des Prozessors auf den gleichen Satz, den zuvor der 8086 genutzt hat; dies gewährt eine Objektcode-Kompatibilität zu bereits vorhandener Software. Für dem Protected Mode hingegen werden zusätzliche Register benötigt.[11]

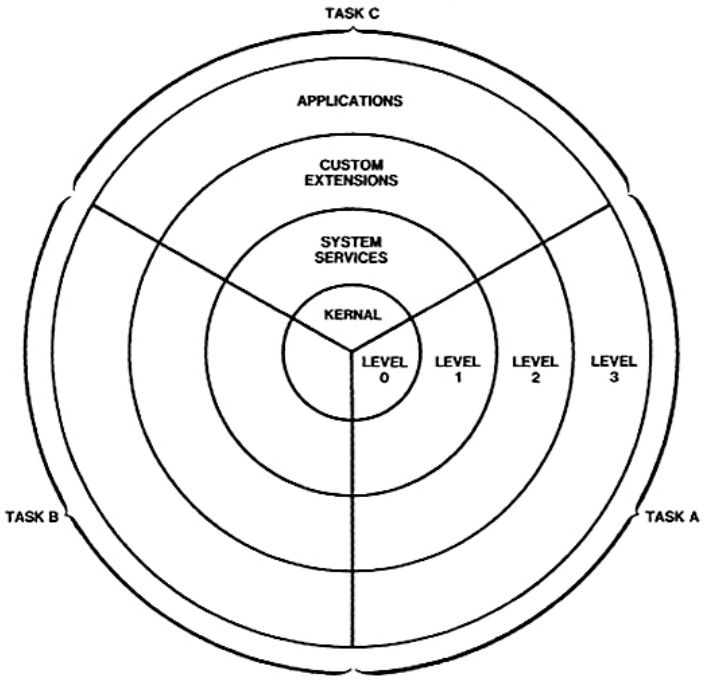

Abb. 5.3: Die 4 Berechtigungsebenen des 80286-Prozessors.

Der 80286-Prozessor bietet im Protected Mode drei wesentliche Schutzmechanismen. Diese umfassen die „Automatische Überprüfung jedes Speicherzugriffs auf Verträglichkeit“, die „Isolierung der verschiedenen Taskbereiche gegeneinander bei Multitasking-Betrieb“ sowie der „Schutz der Betriebssystem-Software vor den Anwenderprorammen (Privileg-Konzept)“[12]. Wird im Ablauf ein Fehler, wie zum Beispiel ein unautorisierter Zugriff auf eine Speicheradresse erkannt, kommt es zu einem Software-Interrupt und das laufende Programm wird unverzüglich gestoppt. In der Architektur des 80286-Prozessors ist es vorgesehen, dass innerhalb vierer Privilegstufen

[10] Vgl.: Thies, Klaus-Dieter: *Das 80186 Handbuch*. Düsseldorf: SYBEX 1986, S. 319.
[11] Vgl.: Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 307ff.
[12] Ebd., S. 343.

unterschieden wird: „Üblicherweise wird Stufe 0 den Betriebssystemkern, Stufe 1 die typischen Betriebssystemdienste, Stufe 2 die Betriebssystemerweiterungen und Stufe 3 dann die Benutzerprogramme enthalten.“[13]

Für die Äußerung der maschinellen Macht über mögliche Handlungen des Users ist Intels Protected Mode ein zentrales Argument. Denn: „Insgesamt gesehen ergibt sich bei der Verwendung von virtuellen Speichern für den Anwender den Eindruck, als ob der Mikrocomputer einen riesigen […] physikalischen Speicher besitzen würde.“[14], so Rudolf Schief. Demnach kann der Betriebsmodus als Mechanismus des Scheins bezeichnet werden. Die Täuschung ist offensichtlich: In der Wahrnehmung des Users steht diesem ein riesiger physikalischer Speicher zur Verfügung, der in der Realität so aber gar nicht existiert. Um dies kurz am Speicher des 286-Prozessors zu verdeutlichen: Während sich der virtuelle Adressraum hier auf 1 GB beläuft, ist der physikalische Adressraum jedoch nur 1 MB groß. Folgender Einwurf sollte an dieser Stelle außerdem erhoben werden: „Diese Schutzmechanismen sind auf dem Chip des 80286 *hardwaremäßig implementiert*, müssen also nicht vom Programmierer softwaremäßig installiert werden und verursachen keine zeitliche Verzögerung bei der Befehlsausführung.“[15] Im Umkehrschluss bedeutet dies: Die schützenden Mechanismen, die im Protected Mode enthalten sind, können gar nicht erst von einem Programmierer implementiert werden, da sie durch die Existenz der Hardware bereits gegeben sind. Den Betriebsmodus vom Protected hin zum Real Mode zu wechseln, ist dabei nicht unmöglich, wird im Programmierhandbuch von 1987 aber nicht explizit erwähnt. In einem aktuell von Intel herausgegebenen Software-Entwicklerhandbuch heißt es dann in Bezug auf den Real Mode: „This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode [ein weiterer Betriebsmodus, welcher der Implementierung von Funktionen wie Energieverwaltung oder Systemsicherheit dient]).“[16] Ein Umschalten vom Real in den Protected Mode scheint also kein Problem darzustellen. Andersherum wird dies allerdings nicht weiter spezifiziert. Rudolf Schief fasst wie folgt zusammen:

> Nach dem Einschalten des Mikroprozessors 80286 bzw. nach einem Hardware-RESET […], begibt sich der Mikroprozessor in den Real-Modus.

[13] Thies: *Das 80186 Handbuch*, S. 321.
[14] Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 327.
[15] Ebd., S. 343.
[16] Intel Corporation: „Intel® 64 and IA-32 Architectures Software Developer’s Manual. Volume 1: Basic Architecture“, 2017, in: https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol1.pdf (Abrufdatum: 13.12.2017), S. 3-1.

> Dies sind die beiden einzigen Möglichkeiten, um den Real-Modus aufzurufen! Es gibt keinen Befehl, um aus dem Virtuellen Adressmodus in den Realen Adressmodus zu gelangen.[17]

Denn: „Das würde das gesamte Schutzkonzept des 80286 durchbrechen."[18] Das Umschalten der Operationsmodi erfordert also spezielle Kenntnisse und liegt nicht – in Erwartung an eine Benutzerfreundlichkeit – einfach so auf der Hand. Der Protected Mode wird im aktuellen Handbuch Intels als „the native state of the processor"[19] beschrieben. Der Prozessor befinde sich also in seinem *ursprünglichen* oder *eigentlichen* Zustand, wenn der Betriebsmodus in den Protected Mode geschaltet ist. Denn seit Intels 286-Prozessor wird er heute von allen modernen PC-Betriebssystemen verwendet.[20] Intel selbst hob außerdem, besonders in den frühen Jahren, deutlich die Vorteile dieses Operationsmodus' hervor. So heißt es im Programmierhandbuch von 1987:

> Only in this mode can the system designer make use of the advanced architectural features of the 80286: virtual memory support, system-wide protection, and built-in multitasking mechanisms are among the new features provided in this mode of operation.[21]

Zudem würden weitaus mehr Fehlerzustände erkannt und interne Interrupts ausgelöst werden. Auch der erheblich erweiterte physikalische Adressraum und der Hardware Support wird hervorgehoben.[22]

In Bezug auf die sich ausbildenden Hierarchien auf Hardwareebene ist hierbei ausschlaggebend, dass eine Verlagerung der Veränderbarkeit stattgefunden hat. Zu Zeiten des 8086-Prozessors, Vorgängermodell des 286-Prozessors, und früher war es noch möglich, die Speicheradressierung auf der Basis von Software zu regulieren. Mit dem Erscheinen des 286-Prozessors ist diese Eigenschaft allerdings in die Hardware gerückt worden, wo die Regulierungen fortan stattfinden. Somit lässt sich zusammenfassen, das mit der Einführung des 286-Prozessors eine Verlagerung der Veränderbarkeit von der Software- auf die Hardwareebene stattgefunden hat. Kittler betont: „Diese guten alten Zeiten sind unwiderruflich vergangen. Unter Stichworten wie Benutzeroberfläche, Anwenderfreundlichkeit oder auch Datenschutz hat die Industrie den Menschen mittlerweile dazu verdammt, Mensch zu bleiben."[23] Und weiter:

---

[17] Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 323f.
[18] Ebd., S. 325.
[19] Intel Corporation: „Intel® 64 and IA-32 Architectures Software Developer's Manual", S. 3-1.
[20] Vgl.: Ebd., S. 2-1ff.
[21] Intel Corporation: „80286 and 80287 Programmer's Reference Manual", S. 5-1.
[22] Vgl.: Ebd., S. 1-2 und S. 2-25.
[23] Kittler: „Protected Mode", S. 209.

> Unterschiedliche Befehlssätze, unterschiedliche Adressierungsmöglichkeiten, unterschiedliche Registersätze, ja sogar unterschiedliche Befehlsausführungszeiten trennen fortan die Spreu vom Weizen, die Anwender vom Systemdesign. In genau dem Silizium also, auf das die Propheten einer mikroprozessierten Zukunftsdemokratie ihre ganze Hoffnung gesetzt haben, kehrt die elementare Dichotomie moderner Medientechniken wieder.[24]

Kittlers kritische Haltung, „die CPU arbeite[t] vielmehr mit Vorrechten und Verboten, Privilegien und Handicaps“[25] sollte an dieser Stelle nicht ohne weiteres Hinterfragen hingenommen werden. Denn ein Betriebsmodus ganz ohne jegliche Schutzmechanismen bringt in der Tat auch einige Nachteile mit sich. Als Produkt des Menschen ist Software dazu prädeterminiert – man könnte auch sagen ‚programmiert‘ –, Fehler zu enthalten.[26] Die Annahme, ein Computerprogramm laufe stets fehlerfrei, ist also schlichtweg illusorisch und riskiert die Stabilität eines ganzen Computersystems. Denn bereits ein falscher Speicherzugriff auf eine Adresse kann zum Absturz oder zu einem schweren Datenverlust führen. Es soll hier also nicht die bloße Existenz eines Betriebsmodus mit Speicherschutz kritisiert werden, welcher vor der Mitleidenschaft nach dem Absturz eines Prozesses oder vor dem ungewollten Eingreifen durch fremde Referenzen schützt, sondern vielmehr die Projektion eines Scheins auf den User, sowie die Tatsache, dass die Entscheidung für eben diese Regulationen stärker in den Händen der Industrie als in den Händen der User liegt.

Erforderlich wurde der Protected Mode insbesondere durch den Multitasking- und Multiuser-Betrieb. Dieser wurde bereits Mitte der 60er Jahre seit der Einführung des *Compatible Time-sharing System* (CTSS) am MIT praktiziert.[27] Auch mit Intels 80286-Prozessor wurde die Fähigkeit, einen Multitasking- oder Multiuser-Betrieb aufnehmen zu können, wieder aufgegriffen. ‚Multitasking‘ bedeutet in diesem Sinne, dass mehrere Programme gleichzeitig durch denselben Computer ausgeführt werden. Unter ‚Multiuser‘ versteht man weiter die Aufteilung einer einzelnen CPU und

[24] Kittler: „Protected Mode“, S. 213.
[25] Ebd.
[26] Einer aktuellen Studie zufolge belaufen sich die Schäden, die alleine in Deutschland durch die Folgen von Softwarebugs entstehen, jährlich auf Milliardenbeträge im zweistelligen Bereich. Vgl.: Asendorpf, Dirk: „Error“, in: *ZEIT ONLINE*, 26.07.2017. http://www.zeit.de/2017/31/computer-software-fehler-systeme (Abrufdatum: 13.12.2017). Dies ist im Übrigen auch Kittler bewusst: „Dringlicher als das Schreiben von Programmen sind Entwanzen und Warten geworden. An ihnen führt kein Weg vorbei, auch nicht der neuerliche Versuch, fehlertolerante Systeme zu entwickeln, weil die Fehlertoleranz selber ja nicht fehlertolerant eingegeben werden kann.“ Kittler, Friedrich: „Computeranalphabetismus“. In: Matejovski, Dirk; Kittler, Friedrich (Hrsg.): *Literatur im Informationszeitalter*. Frankfurt/New York: Campus Verlag 1996, S. 237-251, hier S. 241.
[27] Vgl.: Levy: *Hackers*, S. 109f.

dementsprechend die Vergabe von Ressourcen an mehrere User. Für den Rechner ist es dabei nicht von Bedeutung, ob die gleichzeitig auszuführenden Programme von einem oder verschiedenen Nutzern initiiert wurden; er behandelt sie lediglich als Prozesse, die es sequenziell zu bearbeiten gilt.[28] Die Von-Neumann-Architektur sieht es vor, dass nicht mehr als ein Befehl zu einem Zeitpunkt durch das Rechenwerk ausgeführt werden kann. Auch heute noch kann pro Prozessorkern immer nur ein Prozess in Arbeit sein.[29] Daher ist es auch im Multitasking- bzw. Multiuser-Betrieb faktisch nicht möglich, mehrere Tasks parallel zu bearbeiten. Vielmehr konkurrieren die einzelnen Prozesse um die Betriebsmittel, sodass die Rechnerkapazitäten in solch einer Weise aufgeteilt werden müssen, dass eine schnelle Verarbeitungsfolge stattfinden kann. Die Gleichzeitigkeit des Prozessierens geht also nicht buchstäblich gleichzeitig vonstatten, sondern besteht vielmehr in einem ständigen Hin- und Herspringen zwischen verschiedenen Tasks:

> ‚Gleichzeitiges' Abarbeiten von zwei Tasks heißt dabei, daß der Mikrocomputer zeitlich in raschem Wechsel von einer zur anderen Task springt und so beide Tasks stückweise parallel abarbeitet, so daß der Benutzer den Eindruck hat, der Mikrocomputer sei für jede Task alleine da.[30]

Das gleichzeitige Ausführen von Operationsfolgen wird in der Informatik auch als *Nebenläufigkeit* bezeichnet, doch der englische Begriff *concurrency* macht noch stärker deutlich, wie die verschiedenen Prozesse um die Ressourcen der CPU konkurrieren.[31] Dem User fällt all dies nicht auf, denn mit fortschreitenden Prozessorgenerationen wird auch die Arbeitsgeschwindigkeit stets gesteigert. Dass die Verteilung der Ressourcen mittlerweile in einer Geschwindigkeit vonstattengeht, die für ihn nicht wahrnehmbar ist, lässt sich leicht am Beispiel eines heutigen Prozessors verdeutlichen: Dieser arbeitet mit einer Geschwindigkeit von etwa 1 bis 3 GHz; dies bedeutet umgerechnet eine Menge von 1 bis 3 Milliarden Berechnungen pro Sekunde.[32] Schlussendlich ermöglicht dieser Umstand überhaupt erst, dass beim User der Eindruck des Multitasking entstehen kann. Es bietet sich auch an, hierbei von einer *Pseudo*nebenläufigkeit zu sprechen.[33]

---

[28] Vgl. hierzu: Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 327f. Sowie: Coy, Wolfgang: *Aufbau und Arbeitsweise von Rechenanlagen. Eine Einführung in Rechnerarchitektur und Rechnerorganisation für das Grundstudium der Informatik*. Braunschweig/Wiesbaden: Vieweg 1992, S. 266.

[29] Vgl.: Fischer; Hofer: *Lexikon der Informatik*, S. 711.

[30] Schief: *Einführung in die Mikroprozessoren und Mikrocomputer*, S. 327f.

[31] Vgl.: Vogt, Carsten: *Nebenläufige Programmierung. Ein Arbeitsbuch mit UNIX/Linux und Java*. München: Hanser 2012, S. 25.

[32] Vgl.: Rau, Thomas: „Test: Das bringt ein schneller Prozessor", in: *PC-WELT Online*, 04.10.2012. https://www.pcwelt.de/produkte/Dual-und-Quad-Core-Test-Das-bringt-ein-schneller-Prozessor-6699326.html (Abrufdatum: 16.03.2018).

[33] Diesen Begriff verwendet auch Carsten Vogt in Abgrenzung zur echten Nebenläufigkeit. Vgl. dazu: Vogt: *Nebenläufige Programmierung*, S. 25.

Diese Aufteilung der vorhandenen Ressourcen der CPU setzt gewisse Schutzmechanismen voraus, wie Thies dies 1986 festgehalten hat:

> So ist es in Mehrbenutzersystemen erforderlich, daß Programme und Daten der einzelnen Anwender eindeutig voneinander getrennt sind, ebenso wie auch das Betriebssystem vor Benutzerprogrammen geschützt sein muß. Um jedem einzelnen Benutzer die Möglichkeit zu geben, seine Software unabhängig von der Anzahl der anderen Benutzer zu implementieren und ihm auch den Eindruck zu geben, daß der Rechner nur für ihn allein da ist, ist es notwendig, die CPU durch Multitasking auf die einzelnen Programme aufzuteilen, was dem Benutzer allerdings nur dann verborgen bleiben kann, wenn die CPU sehr leistungsstark ist.
> Außerdem muß jedem Programm ein eigener abgeschlossener virtueller Adreßraum zur Verfügung gestellt werden. Die Hardware muß die Abbildung dieser virtuellen Adressen auf reale Arbeitsspeicheradressen unsichtbar für die Software vornehmen.[34]

Wie Thies' Sprachwahl hier verdeutlicht, soll der *Eindruck* entstehen, als sei der Rechner nur für den Nutzer alleine da, was diesem wiederum *verborgen* bleiben soll. Andererseits sollen die Adressübersetzungsmechanismen auch für die Software *unsichtbar* vorgenommen werden. Insbesondere im Hinblick auf die Ausdrucksweise ist dieses Zitat interessant, denn es lässt sich daran eine weitere Einschreibung der Scheinmechanismen ausmachen. Dass es sich um nicht mehr als die bloße Erzeugung von Illusionen handelt, bestätigt auch Kittler. Mit seinen Worten lässt sich schließen: „Multitasking soll wie der Igel im Märchen den Benutzern vorspiegeln, daß nur ein einziger Igel oder Prozeß läuft, vor allem aber, daß dieser Lauf oder Prozeß auch nur einem einzigen Hasen oder Benutzer zugute kommt."[35] Was Kittler also damit meint, wenn er davon spricht, dass „[u]nter den postmodernen Strategien des Scheins […] keine so wirksam [ist] wie die Simulation, daß es Software überhaupt gibt."[36], ist, dass der Computernutzer daran gehindert wird, die Funktionsmechanismen der Maschine vollständig zu beherrschen. Denn indem technische Möglichkeiten, die theoretisch vorhanden sind, verknappt werden, wird der User von der Maschine und – führt man seine These noch weiter – wiederum von deren Produzenten beherrscht.

Es bietet sich an, in Abgrenzung zu Kittlers Standpunkt eine kurze Aktualisierung auf die heutige Computertechnik vorzunehmen. Seit der Einführung des ersten *Dualcore*-Prozessors durch Intel im Jahr 2006 hat es sich schnell durchgesetzt, (mindestens) zwei

[34] Thies: *Das 80186 Handbuch*, S. 319.
[35] Kittler: „Protected Mode", S. 212.
[36] Ebd., S. 208.

voneinander unabhängige Prozessorkerne in Personal Computern zu verbauen.[37] Bei den sogenannten Mehrkernprozessoren können tatsächlich zwei oder mehr Tasks parallel bearbeitet werden. Was jedoch bleibt, ist die Existenz des Protected Modes. Dieser tritt heute in der Gestalt des sogenannten *Benutzermodus*, auch User Mode, auf und grenzt sich damit vom *Kernelmodus*, auch Supervisor Mode, ab.[38] In Verbindung damit findet auch die virtuelle Speichertechnik eine Fortführung als gebräuchliche Form der Verwaltung von Speicher.[39]

5.2 Beispiel: Sicherheitslücke *Meltdown*

Es mag sich für einige User nicht wie ein direkter Eingriff in ihre Freiheit oder das Feld ihrer Handlungsmöglichkeiten anfühlen, wenn sie mit der bloßen Existenz des Protected Modes auf ihrem PC konfrontiert werden. Sehr viel konkreter wird das Problem, wenn durch die Hardware-Architektur, die grundlegend durch die Aufrechterhaltung eines Speicherschutzes gestaltet ist, Sicherheitsmängel real werden – so geschehen erst kürzlich, als die Prozessor-Sicherheitslücken *Meltdown* und *Spectre* durch die Medien publik geworden sind. Der Vorfall soll an dieser Stelle kurz angeführt werden, da es sich um eine äußerst aktuelle Problematik handelt, die weitreichende Folgen mit sich bringt; hierbei soll vorrangig die Lücke betrachtet werden, die durch den Meltdown-Angriff veranschaulicht wird.

Das Defizit, das später die Bezeichnung *Meltdown* erhielt,[40] wurde unabhängig voneinander von drei Forscherteams, stammend aus Googles *Projekt Zero*, dem deutschen Software-Startup *Cyberus Technology* sowie der Technischen Universität Graz, entdeckt. Innerhalb der Forschergemeinschaft waren die Sicherheitsrisiken seit einigen Monaten bekannt und die Informationen bereits an die verantwortlichen Hersteller weitergeleitet. An die Öffentlichkeit trat die Gruppe am 3. Januar 2018 mit der Bekanntgabe, dass in weiten Teilen der Prozessoren, die in Personal Computern, Tablets

---

[37] Vgl.: Hülsbömer, Simon: „Der x86-Prozessor wird 30 – wie Intel dank IBM alle Gipfel stürmte“, in: *COMPUTERWOCHE Online*, 23.06.2008. https://www.computerwoche.de/a/der-x86-prozessor-wird-30-wie-intel-dank-ibm-alle-gipfel-stuermte,1866928,6 (Abrufdatum: 14.02.2018).

[38] Vgl.: Mandl, Peter: *Grundkurs Betriebssysteme. Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung*. Wiesbaden: Springer Vieweg 2014, S. 25f.

[39] Vgl.: Ebd., S. 221ff.

[40] Der Name *Meltdown* spiegelt die Fähigkeit wider, die Sicherheitsgrenzen, die normalerweise von der Hardware aufrechterhalten werden, zerschmelzen zu können. Vgl. hierzu: Technische Universität Graz: „Meltdown and Spectre. Vulnerabilities in modern computers leak passwords and sensitive data“, in: https://spectreattack.com/ (Abrufdatum: 14.02.2018).

und Smartphones verwendet werden, schwerwiegende Sicherheitslücken bestehen. Meltdown steht in engerem Zusammenhang mit *Spectre*, das als weiteres Angriffsszenario zum Teil von den gleichen Forschern demonstriert wurde und auf ähnlichen Faktoren der Hardwarestruktur in Prozessoren beruht.[41] Auf einer eigens angelegten Website für beide Phänomene durch die Technische Universität Graz heißt es grundsätzlich:

> These hardware vulnerabilities allow programs to steal data which is currently processed on the computer. While programs are typically not permitted to read data from other programs, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs.[42]

Um auf den Speicher fremder Prozesse zugreifen zu können, nutzen die beiden Angriffsszenarien die Eigenschaften der sogenannten *Out-of-order execution* aus. Mit dem Ziel, eine Beschleunigung des Programmablaufs zu erreichen, werden bei diesem Verfahren spekulativ und im Voraus Berechnungen für die nächsten Schritte innerhalb einer Befehlskette angestellt, die im realen Programmfluss so womöglich gar nicht eintreten werden. Dazu werden die vermutlich benötigten Daten in den Cache geladen.[43] Durch eben diese spekulative Befehlsverarbeitung wird Angreifern ein Raum geboten. Denn:

> ‚Aus Performancegründen wird dafür noch nicht überprüft, ob das zugreifende Programm überhaupt die Rechte für einen Zugriff hat', erklären die Grazer Forschenden. Wird der Arbeitsschritt doch nicht benötigt oder fehlen die Zugriffsrechte, dann verwirft der Prozessor die Vorarbeit wieder. Diese Vorarbeit wird bei den neuen Angriffen nun ausgenutzt, um sensible Daten aus dem Kernel auszulesen – beispielsweise Passwörter, die in gängigen Internet-Browsern gespeichert sind.[44]

Grundlage für den Missbrauch der Out-of-order execution ist die Trennung des Supervisor und User Modes. Denn wie bereits erläutert, wurde mit dieser auch die Einführung eines Speicherschutzes praktikabel; folglich hat sich auch die Trennung eines physikalischen und virtuellen Adressraums durchgesetzt. Um die beiden Modi voneinander isolieren zu können, finden Übersetzungsmechanismen der virtuellen auf physikalische Adressen statt, wobei Seitentabellen die Schutz- bzw. Zugriffs-eigenschaften der einzelnen Adressen definieren. Die aktuell genutzte Seitentabelle wird

---

[41] Vgl.: Technische Universität Graz: „Meltdown and Spectre".
[42] Ebd.
[43] Vgl.: Lipp, Moritz [u.a.]: „Meltdown", in: https://meltdownattack.com/meltdown.pdf (Abrufdatum: 14.02.2018), S. 2f.
[44] Baustädter, Birgit: „TU Graz-Forscher entdecken schwerwiegende IT-Sicherheitslücke", in: *TU Graz news*, 04.01.2018. https://www.tugraz.at/tu-graz/services/news-stories/medienservice/einzelansicht/article/schwere-sicherheitsluecke-tu-graz-forscher-zentral-beteiligt/ (Abrufdatum: 16.02.2018).

stets in einem Register festgehalten und mit der Ausführung jedes neuen Prozesses aktualisiert. Während jeder virtuelle Adressraum in einen User- und einen Kernel-Part unterteilt ist, kann auf den Adressraum des Systemkerns nur im Supervisor Mode zugegriffen werden. Da letzterer allerdings nicht nur Speicher für die eigene Verwendung hat, sondern auch Operationen auf Seiten des Users ausführt, wird in der Regel der gesamte physikalische Speicher im Kernel abgebildet.[45] In Verbindung mit der Out-of-order execution ergibt sich daraus die Sicherheitslücke Meltdown:

> Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region.[46]

Das zugrundeliegende Problem geht darauf zurück, dass Prozessoren, die die Fähigkeit zur Out-of-order execution besitzen, es einem unprivilegierten (User-)Prozess erlauben, Daten von einer privilegierten (Kernel- oder physikalischen) Adresse in ein temporäres Register zu laden.

Der Ablauf des Angriffs lässt sich grundlegend in zwei Abschnitte unterteilen. Zunächst wird die CPU dazu gebracht, eine oder mehr Instruktionen auszuführen, die in einem auszuführenden Pfad normalerweise nicht vorkommen würden. Die Autoren Lipp u.a. nennen eine solche Instruktion, die im Zuge der Out-of-order execution ausgeführt wird und dabei messbare Nebenwirkungen hinterlässt, einen ‚transienten Befehl'. Ein solcher hat die ursprünglich nebensächliche, für die Angreifer aber ganz zentrale Wirkung, dass Inhalt von einem Ort des Hauptspeichers, der nicht zugänglich ist, in ein Register geladen werden kann, indem die Daten des Hauptspeichers mithilfe von virtuellen Adressen referenziert werden. Ein transienter Befehl kann dann basierend auf den Daten des Registers auf eine Cache-Zeile zugreifen.[47]

> Although the instructions executed out of order do not have any visible architectural effect on registers or memory, they have microarchitectural side effects. During the out-of-order execution, the referenced memory is fetched into a register and is also stored in the cache. […] This shows that even instructions which are never actually executed, change the microarchitectural state of the CPU.[48]

In einem zweiten Schritt muss dieser mikroarchitektonische Effekt, der durch den transienten Befehl oder die transiente Befehlsfolge entsteht, in einen architektonischen

[45] Vgl.: Lipp [u.a]: „Meltdown", S. 2ff.
[46] Ebd., S. 1.
[47] Vgl.: Ebd., S. 6ff.
[48] Ebd., S. 5.

Zustand übertragen werden, sodass die zwischengespeicherten Informationen weiter prozessiert und letztendlich nach außen geleakt werden können. Um den Status des Caches zu übertragen, bedienen sich die potentiellen Angreifer einer Seitenkanalattacke.[49] Es können also Speicherzugriffe auf den Cache beeinflusst werden, die dann wiederum durch einen Seitenkanal von einem Angreifer in Erfahrung gebracht werden können. Mit dem Ergebnis: „As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known.“[50]

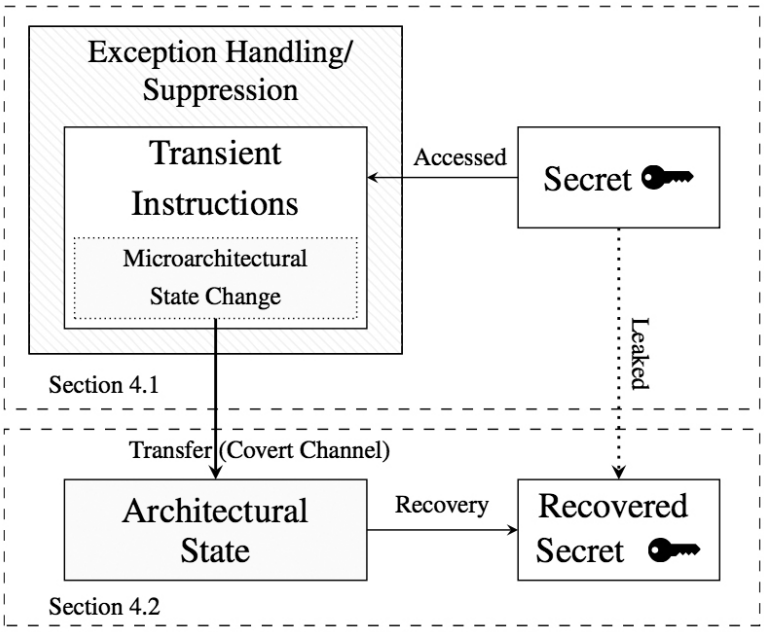

Abb. 5.4: Die beiden Schritte des Meltdown-Angriffs veranschaulicht: Es wird eine Reihe von transienten Befehlen ausgeführt, die zu einer Veränderung des mikroarchitektonischen Zustands des Prozessors führen (Section 4.1). Mithilfe eines verdeckten Kanals wird dieser gelesen und in einen architektonischen Zustand verwandelt. Die geheimen Daten können nun geleakt werden (Section 4.2).

Es ist ein zentraler Faktor, dass der Ursprung des Risikos ausschließlich in der Hardware liegt, denn entsprechend weitreichend sind die möglichen Folgen: „The attack is independant of the operating system, and it does not rely on any software vulnerabilities […], affecting millions of customers and virtually every user of a personal computer.“[51] Wie mittlerweile bekannt ist, sind nicht nur Intel-Prozessoren von der Sicherheitslücke betroffen, sondern auch die Prozessoren einiger anderer Hersteller, darunter

[49] Vgl.: Lipp [u.a]: „Meltdown“, S. 6ff.
[50] Ebd., S. 11.
[51] Ebd., S. 1.

beispielsweise AMD.[52] Auf der zugehörigen Website wird die Frage nach: „Am I affected by the vulnerability?“ mit: „Most certainly, yes.“[53] beantwortet. Die Forschergemeinschaft hat ihre Untersuchungen allerdings nicht ohne mögliche Lösungsvorschläge abgeschlossen. Um dem Hardwaredefizit konsequent entgehen zu können, müsste entsprechend mit Hardwarelösungen reagiert werden. Die offensichtlichste Gegenmaßnahme bestünde hier in der völligen Deaktivierung der Out-of-order execution in den Prozessoren. Aufgrund der Leistungseinbußen, die in Folge dessen allerdings entstehen würden, handelt es sich dabei um keine praktikable Lösung. Ein realistischerer Ansatz wäre es, eine vollständige Trennung zwischen dem Adressraum des Users und dem Adressraum des Kernels einzuführen. Denn dadurch könnte sofort erkannt werden, ob ein Speicherabruf autorisiert oder aber nicht berechtigt ist, und eine Sicherheitsgrenze könnte demnach stets aufrechterhalten werden. Da eine Korrektur der Hardware aber nicht in Kürze allumfassend durchgeführt werden kann, bedarf es einer softwareseitigen Übergangslösung, die in der Kernel-Modifizierung *KAISER* bestehen kann. Ursprünglich wurde diese dazu entwickelt, Seitenkanalattacken abzuwehren, jedoch eignet sie sich auch zur grundlegenden, wenn auch nicht vollkommen sicheren Abwendung eines Meltdown-Angriffs. Mit der Implementierung von KAISER wird verhindert, dass der Kernel-Adressraum in den Adressraum eines Prozesses abgebildet wird, wenn sich der Prozessor im User Mode befindet. Das bedeutet, dass mit dem Wechsel vom User in den Supervisor Mode auch der entsprechende Adressraum gewechselt werden muss.[54]

Abschließend lässt sich festhalten, dass es mit KAISER zwar eine Zwischenlösung zur Abwendung des Sicherheitsrisikos zu geben scheint, diese jedoch erstens nicht als absolut sicher gilt und die Endkunden zweitens darauf angewiesen sind, dass die entsprechenden Updates der Betriebssysteme von den Herstellern zeitnah zur Verfügung gestellt werden. Langfristig muss also über die Softwarelösung hinaus gedacht werden:

> Das vom US-Heimatschutzministerium finanzierte Computer Emergency Response Team (CERT) der Carnegie Mellon University in Pittsburgh ist sogar der Meinung, dass eine wirkungsvolle und nachhaltige Beseitigung der

[52] Vgl.: Windeck, Christof: „Meltdown und Spectre im Überblick: Grundlagen, Auswirkungen und Praxistipps“, in: *heise online*, 19.01.2018. https://www.heise.de/security/meldung/Meltdown-und-Spectre-im-Ueberblick-Grundlagen-Auswirkungen-und-Praxistipps-3944915.html (Abrufdatum: 18.02.2018).
[53] Vgl.: Technische Universität Graz: „Meltdown and Spectre“.
[54] Vgl.: Lipp [u.a.]: „Meltdown“, S. 14.

> geschilderten Cyber-Security-Risiken nur durch den Einsatz neuer Prozessoren möglich ist.[55]

Bis diese Forderung vollends in die Tat umgesetzt werden kann, wird vermutlich eine längere Zeit verstreichen. Dabei ist eine zügige Absicherung äußerst dringlich, ergeben sich bei der Sicherheitslücke doch Angriffsmöglichkeiten von gänzlich neuem Ausmaß: „Meltdown changes the situation entirely. [It] shifts the granularity from a comparably low spatial and temporal granularity, e.g., 64-bytes every few hundred cycles for cache attacks, to an arbitrary granularity, allowing an attacker to read every single bit."[56] So gelang es auch den Autoren um Lipp im demonstrierten Beispielangriff, durch das mehrmalige Wiederholen des beschriebenen Vorgangs auf den gesamten physikalischen Speicher des betroffenen Systems zuzugreifen. Paradoxerweise stellt Meltdown eine Inversion dar: Die Maßnahmen, die ursprünglich zur Aufrechterhaltung einer abgesicherten Ausführung von Programmen entwickelt wurden, werden nun dazu umgekehrt, eben diesen Speicherschutz auszunutzen und eine Sicherheitslücke zu eröffnen. Auch Lipp u.a. halten fest, dass „[…] this hardware-based isolation through a permission bit is considered secure and recommended by the hardware vendors. Hence, modern operating systems always map the entire kernel into the virtual address space of every user process."[57] Und eröffnen damit jedoch eine potentielle Angriffsfläche für Widersacher. Auch wird – wie bereits in Bezug auf den Protected Mode festgestellt wurde – hier noch einmal die starke Abhängigkeit von der Hardwarekomponente deutlich: „Meltdown also shows that even error-free software, which is explicitly written to thwart side-channel attacks, is not secure if the design of the underlying hardware is not taken into account."[58] Denn mit der Hardwareoptimierung wird schließlich die Sicherheit von Softwareimplementierungen gefährdet; das Sicherheitsproblem geht hierbei ausschließlich zurück auf die mikroarchitektonische Ebene des Prozessors. Mit Meltdown lassen sich demnach erneut Kittlers „Unabdingbarkeit" und „Vorgängigkeit von Hardware"[59] unterstreichen.

---

[55] Kroll, Joachim: „Sicherheitslücke in Prozessoren. Was hinter ‚Meltdown' und ‚Spectre' steckt", in: *Computer&AUTOMATION Online*, 08.01.2018. http://www.computer-automation.de/steuerungsebene/safety-security/artikel/149237/ (Abrufdatum: 14.02.2018).
[56] Lipp [u.a.]: „Meltdown", S. 15.
[57] Ebd., S. 8.
[58] Ebd., S. 15.
[59] Kittler: „Es gibt keine Software", S. 237.

6 Machtverhältnisse III – Bedienung von Software

Zuletzt soll die grundlegende These von der maschinellen Macht über Handlungsmöglichkeiten, die vom Computer ausgeht, noch in Bezug auf die Bedienung von Software begriffen werden. Damit wird die Synthese der Software mit der Hardware im Zuge ihrer Programmierung, Ausführung und Bedienung vollständig abgehandelt. Grundsätzlich muss sich dabei die Frage nach der Vermittlung zwischen der Hardware und der *Wetware*[1] durch das Interface bzw. die Benutzeroberfläche gestellt werden. Während die beiden vorangegangenen Kapitel besonders auf die (Un-)Freiheit der Software eingegangen sind, tritt nun stärker ihre Nutzung und damit auch konkret die Stellung des Users in den Vordergrund.

6.1 Benutzeroberfläche

Wie so häufig in der Informatik werden die Begriffe auch in Bezug auf die Benutzeroberfläche oftmals uneinheitlich verwendet. Zu nennen ist in diesem Kontext einerseits die Schnittstelle *Interface*, die sowohl als physikalische und elektronische Kontaktstelle zur Kommunikation von Daten fungiert, als auch die Interaktionsebene in einem geschichteten Kommunikationsmodell, also zwischen den beiden Schichten Mensch und Maschine, beschreibt.[2] Oder kurzgefasst: „In computing, interfaces link software and hardware to each other and to their human users or other sources of data."[3] Andererseits muss auch der Begriff der *Benutzeroberfläche* berücksichtigt werden, der in der folgenden Ausführung bevorzugt verwendet werden soll. Er besagt nämlich grundsätzlich, dass zwischen der Ebene der Maschine und der Ebene des Users eine Oberfläche geschaffen wird, die letzterem zur *Benutzung*, also zur Handhabung von Computerprogrammen, dient. Ein Lexikon der Informatik definiert die Benutzeroberfläche als „Software-Schicht über dem Funktionskern eines Betriebssystems oder einer Anwendung", die „die Interaktions-Schnittstelle zwischen der Benutzerin und dem Kern des entsprechenden Programms" bildet.[4] Sie kann also

[1] Der Begriff meint den menschlichen Anteil, der den Faktoren Hardware und Software beiwohnt. Er nimmt Bezug auf ihre Eigenschaften *hard* und *soft* und spielt darauf an, dass der Mensch zu einem Großteil aus Wasser besteht. Vgl. hierzu: Fischer; Hofer: *Lexikon der Informatik*, S. 997.
[2] Vgl.: Ebd., S. 791.
[3] Cramer, Florian; Fuller, Matthew: „Interface", in: Fuller, Matthew (Hrsg.): *Software Studies. A Lexicon*. Cambridge/London: MIT Press 2008, S. 149-152, hier S. 149.
[4] Fischer; Hofer: *Lexikon der Informatik*, S. 102. Genau genommen verwenden die Autoren den Begriff der Benutz*ungs*oberfläche. Dieser ist jedoch kaum verbreitet; üblicher ist der Begriff Benutz*er*oberfläche.

beispielweise im Desktop mit seinen figurativen Elementen oder in den Dialogschnittstellen bestehen, die das Laden von Applikationen aller Art oder die Dateiverwaltung regeln. Sobald sie auf Symbolen und Fenstern beruht, die mit einer Maus angesteuert werden können, handelt es sich um eine grafische Benutzeroberfläche, auch *Graphical User Interface* (GUI) genannt. Findet die Interaktion dagegen auf einer rein textlichen Basis statt, handelt es sich bei der Oberfläche um eine Kommandozeile, auch *Command Line Interface* (CLI) genannt.[5] Gewissermaßen lässt sich demnach eine Gegenüberstellung von GUI und CLI feststellen, basieren sie beide auf grundlegend anderen Vermittlungswerten. Während ersteres mit seinen metaphorischen Icons in der Tat sehr oberflächlich erscheint, handelt es sich aber auch bei der Kommandozeile um eine programmierte Oberfläche, die den Zugang des Nutzers zum Betriebssystem und zu Anwendungen reguliert.

Für Kittler findet in der Benutzeroberfläche, insbesondere aber in der grafischen Benutzeroberfläche, eine weitere Einschreibung dessen statt, was den User gegenüber dem System abkapselt und ihn deshalb unmündig macht. Er fasst all dies mit dem Begriff des „Computeranalphabeten“ zusammen:

> Den Computeranalphabeten, die Codes weder lesen noch schreiben können, soll dadurch geholfen werden, daß sie mit binären Zahlen und unverständlichen Buchstabenfolgen überhaupt nicht mehr in Berührung kommen. Die Innereien der Maschine bleiben selbstredend weiter digital, weil sie sonst gar nicht laufen würde, aber ihre Benutzerschnittstelle nimmt mehr und mehr die Züge analoger Unterhaltungsmedien an, wie sie seit gut hundert Jahren vertraut sind. Unter dem Schlachtruf Multimedia wird es alsbald eine Neuauflage von Grammophon – Film – Schreibmaschine geben, bei der die Schreib-Rechen-Maschine namens Computer ihre Benutzer nurmehr als analphabetische Augen und Ohren addressiert.[6]

Was Kittler in den 90er Jahren prophezeit hat, ist in der gegenwärtigen Zeit längst Realität geworden. Für die Softwareentwicklung ist die *Usability* oder auch die Benutzerfreundlichkeit, also die Minimierung des Aufwands, den die Benutzung eines Programms für den User erzeugt, heute ein äußerst bedeutender Faktor; Softwareprodukte sollen möglichst ‚ergonomisch‘ entwickelt werden:[7] „Usability, coherence of user

[5] Vgl.: Fischer; Hofer: *Lexikon der Informatik*, S. 102f und S. 168f.
[6] Kittler: „Computeranalphabetismus“, S. 245.
[7] Auch der Begriff von einer Software-Ergonomie ist gebräuchlich: „Ziel der Software-Ergonomie ist die Anpassung der Eigenschaften von Software an die psychischen Eigenschaften der damit arbeitenden Menschen.“ Rundnagel, Regine: „Software-Ergonomie und Benutzungsfreundlichkeit“, in: *ergo-online*, http://www.ergo-online.de/site.aspx?url=html/software/grundlagen_der_software_ergon/software_ergonomie.htm (Abrufdatum: 03.03.2018).

interface, design have become as important in software and computer hardware as they are in consumer electronics."[8] Dass die Thematik ein höheres Maß an Komplexität erreicht hat, kann daran veranschaulicht werden, dass innerhalb des GUI nunmehr zwischen dem *User Interface* (UI) und der *User Experience* (UX) unterschieden wird. Während ersteres in der reinen Gestaltung der Oberfläche besteht, umfasst letzteres das Anwendererlebnis – hier geht es darum, Informationen architektonisch so anzuordnen, dass sie intuitiv bedient werden können:[9] „Anwendungen, die über eine ‚intuitive' Bedienung […] verfügen, weisen automatisch ein gutes Anwendererlebnis auf – der Nutzer fühlt sich einfach wohl[,] während er das Interface nutzt."[10] Es wird also versucht, dem User die Bedienung so unkompliziert und bequem wie nur möglich einzurichten, sodass sich bei diesem ein ‚Wohlgefühl' äußern kann.

Dass das Streben nach einer allseitigen Benutzerfreundlichkeit auch eine Problematik mit sich bringt, lässt sich treffend mit den Worten Frank Hartmanns sagen, der die Konsequenz aus der Angelegenheit zieht: „Computer sind einfacher zu bedienen, aber umso schwerer bleibt zu verstehen, was sich hinter den Benutzeroberflächen ‚eigentlich' noch verbirgt."[11] An der Benutzeroberfläche entlang verläuft also eine Kluft, die das für den User Sichtbare klar von dem für ihn Unsichtbaren trennt. Die Kommunikationswege zwischen ihm und der Maschine verlaufen also verkürzt, indem der optische Kanal eines Bildschirms (sowie der akustische Kanal eines Lautsprechers) zwischengeschaltet ist, welcher Augen (und Ohren) des Computernutzers adressiert.[12] Dadurch können Befehlsmöglichkeiten und Daten allerdings nicht bloß offengelegt, sondern andererseits auch verdeckt und verborgen werden. Was hinter den Kulissen, oder genauer gesagt unter seiner Oberfläche geschieht, kann der Nutzer, der in diesem Sinne auch ein bloßer Betrachter ist, nicht nachvollziehen. So stellt auch Pias in Bezug auf den Computerspieler – aber mit allgemeiner Gültigkeit – fest:

> In die Mitte rückt vielmehr das Medium eines Interface, das zwischen dem Widerspruch von Menschen und Maschinen, hardware und wetware, vermittelt und dabei zugleich erst erzeugt und formatiert, was ein Mensch als

[8] Pfeiffer Consulting: „User Interface Friction Research. Research Report", in: http://www.pfeiffer report.com/v2/wp-content/uploads/2013/05/UIF_Rep.pdf (Abrufdatum: 26.02.2018), S. 14.
[9] Vgl.: Janschitz, Mario: „UI vs UX: ‚So sieht es aus' vs. ‚So fühlt es sich an'", in: *t3n Online*, 03.09.2015. https://t3n.de/news/ui-ux-sieht-vs-fuehlt-635734/ (Abrufdatum: 26.02.2018).
[10] Ebd.
[11] Hartmann, Frank: „Vom Sündenfall der Software", in: *Telepolis*, 22.12.1998. https://www.heise.de/tp/features/Vom-Suendenfall-der-Software-3601396.html (Abrufdatum: 04.10.2017).
[12] Vgl.: Krajewski: „Dienstleistungsagenturen", S. 128.

> User ist. Am Interface verhalten sich Spieler nicht nur, sondern werden auch verhalten.[13]

Schließlich wurde eine absichtlich oberflächliche Ebene erzeugt, mithilfe derer die mögliche Autonomie des Nutzers – im Sinne einer wahren Interaktivität mit dem Computer – durch den Softwaredesigner untergraben wird.[14] Durch Benutzerschnittstellen, deren Design sich an den Unterhaltungsmedien orientiert, wird der User erheblich auf Distanz gehalten. Kittlers „Märchen von einer Softwareentwicklung, die schon immer sanfter und benutzerfreundlicher, spiritueller und intelligenter geworden wäre,“[15] findet damit auch in der Bedienung von Software eine Fortsetzung.

6.2 *User Interface Friction*

> Zunächst einmal wurden, auf einer mit Absicht oberflächlichen Ebene, brauchbare graphische Schnittstellen entwickelt, die, weil sie die zur Programmierung immer noch unumgänglichen Schreibakte verstecken, eine ganze Maschine ihren Benutzern entziehen. Denn nicht einmal das IBM-autorisierte Computergraphikkompendium gibt vor, daß computergraphische Benutzeroberflächen die Systemprogrammierung schneller oder effizienter als schlichte Kommandozeilen machen würden.[16]

Dass Benutzeroberflächen, die dazu dienen, die Macht über die Steuerung des Computers den Programmierern und nicht den Usern zuzusprechen, zudem oftmals keine Optimierung der Benutzung bezwecken, macht die sogenannte *User Interface Friction* deutlich. Der Begriff beschreibt konkret die Lücke, die zwischen dem Arbeitsfluss des Users und der Benutzeroberfläche entsteht, insbesondere dann, wenn Software schlecht implementiert ist und die Funktionalität der Maschine Einschränkungen erfährt. Das Problem kann in Analogie zum Schreiben mit Stift und Papier veranschaulicht werden: „Just like the smoothness of the paper or the ink-flow of a pen can impact the speed of handwriting, User Interface Friction affects practically any procedure where the user interacts through the user interface with the computer system.“[17] Meist beziehen sich die negativen Auswirkungen hierbei auf eine Verlangsamung der Rechenleistung. So kann die Wartezeit darauf, dass sich ein Menü oder Untermenü öffnet, in messbaren Produktivitätsverlusten resultieren, indem der User ausbremst wird. Auch die Bedienung

---

[13] Pias, Claus: „Die Pflichten des Spielers. Der User als Gestalt der Anschlüsse“, in: Warnke, Martin; Coy, Wolfgang; Tholen, Georg C. (Hrsg.): *HyperKult II. Zur Ortsbestimmung analoger und digitaler Medien*. Bielefeld: transcript 2005, S. 313-341, hier S. 315.
[14] Vgl.: Hartmann: „Vom Sündenfall der Software“.
[15] Kittler: „Protected Mode“, S. 208.
[16] Kittler: „Es gibt keine Software“, S. 233.
[17] Pfeiffer Consulting: „User Interface Friction Research“, S. 26.

einer Maus kann seine Effizienz beeinträchtigen, insbesondere wenn die gerade zu erledigende Aufgabe eine gewisse Präzision erfordert, denn dann geht auch hier eine Verzögerung als Wirkung hervor. Ähnlich verhält es sich mit nicht eindeutig klaren Symbolen oder einer mehrdeutigen Darlegung der Optionen, die beim User Zögern und Verzögerung hervorrufen. Ein weiterer, oft als problematisch angesehener Umstand ist es, dass es für den User mühseliger wird, in dem für moderne Betriebssysteme typischen Modell aus Dateien und Ordner zu navigieren, je mehr die Zahl dieser Dateien und Ordner zunimmt. Ebenfalls führt es zu einer Überforderung desselben, wenn ihm eine Fülle an verfügbaren, unzureichend strukturierten Optionen präsentiert wird, was mehrdeutige Symbole noch verstärken können. Schließlich und übergreifend können auch die Konzepte, die ein visuelles Interface vermitteln will, schlichtweg zur Verwirrung führen, wenn sie immer abstrakter werden.[18]

All dies sind Beispiele für mögliche Faktoren, die eine User Interface Friction hervorrufen können. Es ließen sich sicher noch weitere anführen; auch müssen die genannten Aspekte nicht notwendigerweise negative Auswirkungen mit sich bringen. Schließlich kommt es bei der grafischen Benutzeroberfläche sehr auf Details an, die über ihre Praktikabilität entscheiden: „Two programs that both use pull-down menus offering the same options can vary in efficiency just by the way in which the menus are organized."[19] Auch muss mit Blick auf die Kompetenzen zwischen den Usern unterschieden werden, denn die User Interface Friction ist nicht identisch für den durchschnittlichen Computernutzer und den Experten. Während ersterer oftmals auf eine veranschaulichende Oberfläche zum Verständnis angewiesen ist, rufen die dieselben Verknappungen beim technikversierten Nutzer wohl eher Frustrationen hervor.[20] Fest steht, dass die User Interface Friction insbesondere dann, wenn sie bei häufig wiederholten Vorgängen auftritt, zu einem erheblichen Produktivitätsverlust führt. In erster Linie kommt es im Ablauf dann zu Wartezeiten und Verzögerungen – entweder bei direkter Ausführung einer Aktion oder indirekt, wenn der User aufgrund von Unsicherheiten in seinem Handeln zögert. Die Verlangsamung des Computings steht dabei im starken Gegensatz zu der Weiterentwicklung der Computertechnik, die in schnellen Schritten voranschreitet, um stets eine noch bessere Rechenleistung erreichen zu können. Das Resultat ist, dass: „[E]legant hardware will suffer from an inefficient user

[18] Vgl.: Pfeiffer Consulting: „User Interface Friction Research", S. 27ff.
[19] Ebd., S. 12.
[20] Ebd., S. 14.

interface."[21] Es lässt sich demnach hinterfragen, wieso es einer Schnittstelle bedarf, die in ihrer Art und Weise dem User dabei im Wege steht, die technischen Möglichkeiten der Maschine vollends auszunutzen und die Aktionen entsprechend selbstständig zu lenken. Die Forscher der User Interface Friction vertreten die Meinung, dass es dem modernen Computernutzer nicht wichtig sei, die modernste Funktionalität ausschöpfen zu können. Vielmehr lege er Wert auf einen bequemen und unkomplizierten Gebrauch:

> Any technology is only as good as the part of it that is actually used. While in earlier days of computing, a large number of users may have had technical curiosity, enjoying the discovery and experimentation when a new release of a program or operating system arrived, average computer users today care little about cutting-edge functionality, and are mainly focused on getting their job done as fast as possible.[22]

Dass gerade die Benutzeroberfläche Wartezeiten generiert, wo dem Benutzer doch vor allem daran gelegen ist, möglichst schnell mit dem Computer interagieren zu können, bleibt ein Paradox, auf das an dieser Stelle lediglich hingewiesen werden kann.

Es sollte berücksichtigt werden, dass die Auswirkungen des Verhältnisses zwischen Funktionalität auf der einen und Implementierung auf der anderen Seite auch umgekehrt werden kann. Während eine schlechte Implementierung in vielen Fällen die Leistung des Rechners einschränkt, kann eine sehr gute Implementierung auch für einen Ausgleich einer eher begrenzten Funktionalität auf Seiten der Hardware sorgen, zumindest in Bezug auf die Empfindung eines – durchschnittlichen – Users.[23] Unabhängig davon, wie die Auswirkungen gedeutet werden: Zwischen der Wahrnehmung des Users und den technischen Gegebenheiten besteht eine Diskrepanz. So soll noch auf folgende Frage hingewiesen werden, die bei der Gestaltung der grafischen Benutzeroberfläche aufkommt: „Should the user be able to understand what he or she is doing?"[24] Sie illustriert, wie die Designer der GUI grundlegend darüber entscheiden können, was ein Teil des Verständnisses des Nutzers und was davon auszuschließen ist. Somit gestalten sie nicht bloß das Interface, sondern auch die Wahrnehmung des Menschen. Die User Interface Friction macht schließlich deutlich, dass die Benutzeroberfläche ihrem Namen völlig gerecht wird. Denn sie sorgt dafür, dass die Interaktion zwischen Mensch und Maschine auf einer rein oberflächlichen Ebene existiert.

---

[21] Pfeiffer Consulting: „User Interface Friction Research", S. 31.
[22] Vgl.: Ebd., S. 17, S. 31.
[23] Vgl.: Ebd., S. 25.
[24] Ebd., S. 18.

7 Zur Dialektik von Software und Hardware

Nach der Abhandlung der verschiedenen Aspekte, die eine Vorherrschaft der Hardware deutlich gemacht haben, soll an dieser Stelle ein wenig Abstand von Details genommen werden, sodass ein umfassender Blick auf das komplexe Verhältnis der Software zur Hardware möglich wird.

Eine grundsätzliche Unterscheidung der beiden besteht darin, dass die Hardware im physischen und die Software im symbolischen Teil eines rechnenden Computers vorwaltet. Der Begriff *Hardware* einerseits fasst unter anderem die zentralen Rechnerkomponenten wie zum Beispiel Mainboard, Mikroprozessor und Laufwerke zusammen und macht damit deutlich, dass es sich um feste *Bau*teile handelt, die physisch miteinander verknüpft und deshalb nicht auf die Schnelle veränderbar sind.[1] Der Begriff *Software* andererseits beschreibt neben Daten hauptsächlich Programme und besteht damit grundsätzlich in den Rechen*anweisungen*, die vom Computer ausgeführt werden. Der Gedanke des Symbolischen manifestiert sich hier insofern, als dass sie weder angefasst noch verbaut werden können. Schon in der Gegenüberstellung der Begriffe wird ein Widerspruch deutlich, denn die Hardware weist gegenüber der Software eine greifbare Materialität auf; sie bildet demnach den Ausgangspunkt, der für die Gestaltung der Software maßgeblich ist. Heißt es also Physik vor Symbolik, handelt es sich dabei nicht bloß um eine Kausalkette, denn auch ein Machtverhältnis wird hieran sichtbar: der Umstand, dass die Hardware gegenüber der Software Vorrang hat – wie diese Arbeit bereits mehrmals deutlich gemacht hat. Andererseits, auch dies demonstriert die Begriffsstudie, ist es nach wie vor die Aufgabe der Software, den Computer mit Befehlen zu instruieren. Diese Gegensätzlichkeit war den beiden immer schon gegeben: Blickt man zurück in die Geschichte des Computers, so erkennt man, dass Hardware die Entwicklung von Software erst ermöglicht hat. Umgekehrt jedoch begünstigt auch Software stets die Weiterentwicklung der Maschinerie. Kurzum, die Beziehung der beiden Aspekte zueinander erscheint kompliziert.

[1] Der Begriff *hardware* bedeutet im Englischen ursprünglich schlichtweg ‚Eisenwaren' in einem von dem Computer unabhängigen Sinn. Kittler schreibt dazu: „Denn in jener guten alten Zeit, als auch Engländer und Amerikaner unter Hardware noch das Warensortiment von Eisenwarengeschäften verstanden, schloß der Begriff von Werkzeug den der Programmierbarkeit eher aus. Es gehörte zur (in *Sein und Zeit* zu Recht gerühmten) Verläßlichkeit von Hämmern, sich nicht unter der Hand in Sägen oder Bohrer zu verwandeln." Kittler: „Hardware, das unbekannte Wesen", S. 119f.

Es soll an dieser Stelle die These aufgestellt werden, dass das Verhältnis von Software und Hardware dazu *bestimmt* ist, von einer solchen Dialektik geprägt zu sein. Da der Begriff der *Dialektik* in diesem Kontext nicht einfach mit dem einer Widersprüchlichkeit oder einer Divergenz gleichgesetzt werden soll, erfolgt eine Erläuterung dessen und damit auch ein kurzer Exkurs in die Philosophie. Geprägt wurde der Begriff insbesondere durch Hegel, mit dessen Worten sich die Dialektik im Wesentlichen wie folgt zusammenfassen lässt: „Alle Dinge sind an sich selbst widersprechend."[2] Was heute als die hegelsche Dialektik bekannt ist, nimmt für diesen genau genommen einen Teil der Logik ein. Diese nämlich besteht, so Hegel, aus drei grundlegenden Momenten: „Das *Logische* hat der Form nach drei Seiten: *α) die abstrakte* oder *verständige*, *β) die dialektische* oder *negativ-vernünftige*, *γ) die spekulative* oder *positiv-vernünftige*."[3] Die dialektische bzw. negativ-vernünftige Seite der Logik besteht für ihn in dem „eigene[n] Sichaufheben solcher endlichen Bestimmungen und ihr Übergehen in ihre entgegengesetzten."[4] Dies führt er weiter aus:

> In ihrer eigentümlichen Bestimmtheit ist die Dialektik vielmehr die eigene, wahrhafte Natur der Verstandesbestimmungen, der Dinge und des Endlichen überhaupt. […] Die Dialektik dagegen ist dies *immanente* Hinausgehen, worin die Einseitigkeit und Beschränktheit der Verstandesbestimmungen sich als das, was sie ist, nämlich als ihre Negation darstellt. Alles Endliche ist dies, sich selbst aufzuheben.[5]

Die Einseitigkeiten des Denkens erfahren also prinzipiell eine Selbstaufhebung. Denn die Vernunft erkennt und verneint eine solch einseitige Bestimmung von etwas Seiendem, sodass ein inhärenter Widerspruch entsteht. Da sich die begrifflichen Gegensätze der Dinge einander negieren, heben sie sich als Resultat daraus gegenseitig auf. Der Philosoph de Vos fasst wie folgt zusammen:

> Hegels Gebrauch des Terminus ‚Dialektik' bezieht sich demnach auf das Verhältnis der Vernunft zum Verstand bzw. des Endlichen zum Absoluten; die Dialektik ist das treibende Moment des Vernünftigen innerhalb des Verstandesdenkens, durch das sich der Verstand schließlich selbst aufhebt.[6]

Es handelt sich bei Hegels Dialektik demnach um einen zentralen, die Logik aller Dinge konstituierenden Aspekt. Dies erklärt, wieso er in der Philosophie eine häufige

---

[2] Hegel, Georg Wilhelm Friedrich: *Wissenschaft der Logik II. Werke 6*. Frankfurt am Main: Suhrkamp 1990, S. 74.
[3] Hegel, Georg Wilhelm Friedrich: *Enzyklopädie der philosophischen Wissenschaften I. Werke 8*. Frankfurt am Main: Suhrkamp 1989, S. 168.
[4] Hegel: *Enzyklopädie der philosophischen Wissenschaften I*, S. 172.
[5] Ebd., S. 172f.
[6] de Vos, Lu: „Dialektik", in: Cobben, Paul [u.a.] (Hrsg.): *Hegel-Lexikon*. Darmstadt: Wissenschaftliche Buchgesellschaft 2006, S. 142-143, hier S. 142.

Anwendung erfahren hat; so auch in Bezug auf die Freiheit: Die sogenannte *Dialektik der Freiheit* ist ein Diskurs, der schon seit langer Zeit in der Philosophie vorherrscht und sich grundsätzlich mit der Tatsache auseinandersetzt, dass der Begriff der Freiheit per se durch einen inneren Widerspruch gekennzeichnet ist. Eine intensive Beschäftigung mit der Thematik lässt sich bei Adorno finden; so etwa in seiner Vorlesung *Probleme der Moralphilosophie* von 1956/57. Darin illustriert er die Moral als „Unterdrückung, Repression insofern, als sie den Menschen positiv Freiheit zuspricht und sie für alles zur Verantwortung zieht", aber auch als „Kritik an dem, was die Menschen tun, Repräsentantin einer kommenden Freiheit […]. Daraus folgt, daß die Moral selbst in sich widerspruchsvoll ist insofern, als sie gleichzeitig immer Freiheit und Unterdrückung meint."[7] Schweppenhäuser, der sich im Zuge seiner Dissertation mit Adornos kritischer Theorie auseinandergesetzt hat, fasst die, wie er sie nennt, „heikle[n] Frage" folgendermaßen zusammen: „Einerseits will Adorno zeigen, wie blockiert Freiheit und Selbstbestimmung sind. Andererseits wird den Individuen zugemutet, sich zu verhalten, als sei dies nicht der Fall."[8] So hat bereits Adorno die immanente Einheit von Freiheit und Unfreiheit festgestellt.

„Immer mehr Freiheit! Immer mehr Kontrolle."[9] Dass die Dialektik der Freiheit auch in gegenwärtigen Zeiten weiterhin ein Phänomen von großer Relevanz ist, macht das Autorengespann Metz und Seeßlen deutlich:

> Es gibt so viel zu kontrollieren. Und immer ist ihnen [den Menschen] genau da die Kontrolle so vollständig entglitten, wo sie sich ihrer Vorstellung nahe glaubten. Wenn sie so frei sind, etwas vor sich zu sehen oder sogar in sich, dann können Menschen es nur ertragen, wenn sie es […] auch kontrollieren können.[10]

Denn fest steht, so schreiben sie ferner: „Es gibt keine Freiheit ohne Kontrolle, und es gibt keine Kontrolle ohne Freiheit."[11] Vielmehr geht nämlich eine ständige Oszillation zwischen den beiden Polen von Freiheit und Kontrolle vonstatten. Diese Dialektik geht konsequentermaßen schon damit einher, dass ein Verständnis von Freiheit erst dann

[7] Adorno, Theodor W.: Probleme der Moralphilosophie. Vorlesung, gehalten im Wintersemester 1956/57 an der Universität Frankfurt. Zitiert nach: Schweppenhäuser, Gerhard: *Ethik nach Auschwitz. Adornos negative Moralphilosophie*. Hamburg: Argument-Verlag 1993, S. 178f.
[8] Ebd., S. 180.
[9] Metz, Markus; Seeßlen, Georg: *Freiheit und Kontrolle. Die Geschichte des nicht zu Ende befreiten Sklaven*. Berlin: Suhrkamp 2017, S. 12.
[10] Ebd., S. 11.
[11] Ebd., S. 9.

entstehen kann, wenn bereits ein Maß an Unfreiheit erfahren wurde.[12] Zudem macht die Aufrechterhaltung eines freien Umfeldes die Kontrolle desselben notwendig: „Das einzig wirksame Mittel, Freiheit nachhaltig zu realisieren, vom Rausch der Befreiung zu einem stabilen gesellschaftlichen System zu gelangen, ist – Kontrolle."[13] Dies lässt sich am Beispiel der Demokratie illustrieren; ihr Funktionieren wird dadurch sichergestellt, dass die Macht ausübenden Institutionen wiederum von anderen Institutionen in ihrem Tun überprüft und kontrolliert werden, mit dem Ergebnis, ein freiheitliches politisches System zu erzeugen. Die absolute Freiheit ist demnach auszuschließen:

> Auch Freiheit ist […] ein Versprechen, das sich nur in einem Immer-Mehr erfüllen will, und das sich immer wieder als trügerisch erweist. So wie man die totale Kontrolle nur über Systeme, Gemeinschaften und Menschen erlangen kann, die nicht mehr lebendig sind, totale Kontrolle also den geistigen, seelischen und sogar körperlichen Tod des Kontrollierten bedeuten würde, so würde totale Freiheit eine rücksichtslose gegenseitige Vernichtung bedeuten.[14]

Frank Wiebe, der sich ausgehend von dem Werk des Philosophen Fichte der Thematik nähert, bezeichnet die beiden Seiten von Freiheit auch als den positiven und den negativen Begriff von Freiheit. Während der negative Freiheitsbegriff, also die Freiheit *wovon*, darin existiert, sich etwa von äußeren Zwecken frei zu machen, besteht der positive Freiheitsbegriff, die Freiheit *wozu*, in der Chance, etwas tun zu können.[15] Auch Wiebe macht damit deutlich, dass der innere Widerspruch der Freiheit bzw. der Widerspruch zwischen Freiheit und Kontrolle in einer Dualität mündet, die für die Konstitution beider Konzepte notwendig ist.

Mit dem Begriff der Dialektik tut sich also ein interessantes Spannungsfeld auf, in welches sich auch die Software und die Hardware einbetten lassen. Denn damit können diese – wie zuvor argumentierten – ‚widersprüchlichen' Aspekte gewissermaßen zusammengeführt werden. Die Medientheoretikerin Wendy Chun beispielsweise setzt das Verhältnis von Hardware und Software mit dem Verhältnis von Material und

---

[12] In einer Randnotiz bemerken die Autoren: „Es soll, raunt es uns aus den hinteren Regalen der Bibliothek zu, menschliche Gesellschaften gegeben haben, in denen es kein Wort für Freiheit gab. Ganz einfach deswegen, weil es auch keines für das Gegenteil davon gab." Metz; Seeßlen: *Freiheit und Kontrolle*, S. 13, Anmerkung 1.

[13] Metz, Markus; Seeßlen, Georg: „Über die Freiheit. Die dialektische Beziehung von Freiheit und Kontrolle", in: *Deutschlandfunk Online*, 05.10.2014. http://www.deutschlandfunk.de/ueber-die-freiheit-die-dialektische-beziehung-von-freiheit.1184.de.html?dram:article_id=294982 (Abrufdatum: 12.01.2018).

[14] Ebd.

[15] Vgl.: Wiebe, Frank: „Fichte und die Dialektik der Freiheit", in: *SciLogs*, 13.09.2012. https://scilogs.spektrum.de/gute-geschaefte/fichte-und-die-dialektik-der-freiheit/ (Abrufdatum: 12.01. 2018).

Ideologie gleich: „Software and ideology fit each other perfectly because both try to map the material effects of the immaterial and to posit the immaterial through visible cues.“[16] Es entsteht dadurch, so Chun, ein neues Verständnis von Software, in welchem diese mit einer funktionalen Ideologie gleichzusetzen ist. Denn wie auch die Ideologie ein falsches Bewusstsein von den eigenen Existenzbedingungen erzeugt, die wiederum auf hegemonialen Machtdiskursen beruhen, welche nicht zugänglich sind, behindert auch die Software den Nutzer in seinen Erkenntnissen. In Verbindung mit der Hardware – schließlich werden sie zusammen vom Computer umfasst – entsteht dadurch so etwas wie eine Maschine der Ideologie.[17] Auch Wolfgang Hagen lässt sich an dieser Stelle anführen, denn er beschreibt Software als Stil, der etwas ermöglicht, dasselbe aber gleichzeitig verbirgt. So äußert er sich in seiner Abhandlung *Der Stil der Sourcen* über die Entstehung der Programmiersprachen wie folgt zur Metonymie des Stils: „Unkenntlichmachen des Geschriebenen durch das Geschriebene. […] Das ist der Stil.“[18] Daraus lässt sich schließen, dass die Software eine inhärent dialektische Eigenart aufweist, nämlich das zu verhüllen, was sie selbst hervorgebracht hat.

Die Möglichkeiten für eine Veranschaulichung aus der Medientheorie sind groß, ebenso gut lassen sich aber auch Beispiele aus der Praxis finden, die das dialektische Verhältnis demonstrieren. So etwa das Problem, das daraus entsteht, ein funktionierendes Smartphone zu besitzen, für das kein Update der Betriebssoftware mehr erhältlich ist. Die Hersteller verhindern Updates der Betriebssoftware in der Regel viele Jahre nach Verkaufsstart, wenn sich bereits einige Nachfolger des besagten Geräts auf dem Markt etabliert haben. Daraus entstehen ökonomische Folgen, die direkt für den User spürbar sind: Obwohl das Gerät hardwareseitig intakt ist und noch problemlos funktionieren könnte, wird es durch die fehlende Aktualisierung der Software unbenutzbar: „Der Computer als technisches Gerät, die Hardware allein ohne Software, läßt sich nicht ohne weiteres nützlich verwenden.“[19] So wird der User durch die Industrie zu einem gewissen Grad für unmündig erklärt. Ein weiteres Exempel stellt der Online-Dienst *GitHub* für die Versionsverwaltung dar. Prinzipiell erleichtert dieser die Codeproduktion, insbesondere für Freie und Open-Source-Software, da Entwickler ihren Code dort zur freien Verfügung

[16] Chun, Wendy H. K.: „On Software, or the Persistence of Visual Knowledge“, in: *Grey Room*, Band 18, Heft o. A. (2005), S. 26-51, hier S. 44.
[17] Vgl.: Ebd., S. 42ff.
[18] Hagen: „Der Stil der Sourcen“, S. 39.
[19] Rechenberg, Peter: *Was ist Informatik? Eine allgemeinverständliche Einführung*. München: Hanser 1994, S. 117.

stellen, aber auch zur Diskussion und für Weiterentwicklungen veröffentlichen können. Schließlich ermöglicht GitHub durch den Aspekt des *Social Codings* eine bessere Kommunikation und mehr Produktivität unter Softwareentwicklern.[20] Andererseits aber erfolgt hier auch eine Kontrolle durch das System selbst: Welche Dateienformate gelesen werden können oder welche Dateigrößen erlaubt sind, wird von GitHub klar festgelegt.[21] Somit stehen sich die Ermöglichung und Regulierung des kooperativen Programmierens direkt gegenüber.

Wie sich gezeigt hat, scheint der Begriff der Dialektik besonders gut dafür geeignet zu sein, die wechselseitigen Verbindungen, die zwischen der Software und Hardware, aber auch innerhalb des Diskurses von Software[22] und desgleichen von Hardware bestehen, zu beschreiben. Die Determination der Wechselseitigkeit hilft dabei, das prekäre Verhältnis der beiden Computerbestandteile besser verstehen zu können und dieses nicht weiter infrage zu stellen, indem der offensichtlichen Widersprüchlichkeit Sinn zugesprochen wird. In Bezug auf den Begriff von Freiheit allerdings lässt sich darüber diskutieren, ob nicht bereits eine Auflösung der Dialektik stattgefunden hat, wie Metz und Seeßlen argumentieren:

> Wir haben uns angewöhnt, Freiheit vor allem als Abwesenheit von Kontrolle (falsch) zu verstehen, und umgekehrt Kontrolle vor allem als eine Art Gift für die freie Entfaltung (auf dem Markt). Eine Freiheit auf dem Markt ist letztlich grenzenlos und schließt vor allem die Kontrolle der Konkurrenz beziehungsweise der User mit ein. Freiheit und Kontrolle verhalten sich nicht mehr dialektisch zueinander, sondern entsprechend dem Bild vom Hasen und vom Igel. Wobei möglicherweise noch nicht endgültig geklärt ist, welche Seite die Rolle des Hasen und welche die des Igels übernimmt.[23]

Die Frage nach der Freiheit soll schließlich im abschließenden Kapitel dieser Arbeit dargelegt werden.

---

[20] Vgl.: Thung, Ferdian [u.a.]: „Network Structure of Social Coding in GitHub“, in: *17th European Conference on Software Maintenance and Reengineering (CSMR 2013), März 2013, Genua*, S. 1-4.

[21] Theoretisch sind in einem sogenannten *Repository*, dem zu verwaltenden Verzeichnis, alle Dateiformate erlaubt, allerdings werden sie nicht alle problemlos von der Software verstanden. Vgl. hierzu: Perkel, Jeffrey: „Democratic Databases: Science on GitHub“, in: *Nature*, Band 538, Heft 7623 (2016), S. 127-129, hier S. 128.

[22] Gerade in Bezug auf die Programmierung lässt sich der Dialektik-Begriff auf ihre grundlegendste Basis zurückführen: Denn schon in der Linguistik kann mit dem gegenseitigen Verweis von Signifikant und Signifikat nach Saussure ein dialektisches Verhältnis innerhalb des Zeichens festgestellt werden.

[23] Metz; Seeßlen: „Über die Freiheit“.

8 Fazit

Es folgt ein Zusammentragen der aus der Arbeit gewonnenen Erkenntnisse in Bezug auf die (un-)freie Computernutzung. Zudem werden Überlegungen über mögliche Alternativlösungen angestellt, die von Freier Software deutlichen Abstand nehmen.

8.1 Die Frage nach der Freiheit

Die vorliegende Arbeit ging von der Existenz einer Softwarekategorie aus, die weithin als frei gilt. Anlässlich der gründlichen Darlegung des Diskurses um Freie Software wurde diese grundsätzlich als eine die Freiheit ihrer Nutzer respektierende Software definiert, die sich insbesondere von einer – in diesem Sinne ‚unfreien' – proprietären Software abgrenzt. Ebenso wurde aufgezeigt, dass die zentralen Nutzungsfreiheiten der Freien Software – darin bestehend, ein Programm ausführen, kopieren, weiterverbreiten und verändern zu können – in Lizenzen festgeschrieben werden, die meist dem Prinzip des Copyleft folgen und somit den Quellcode eines Programms mit den genannten Freiheiten verketten. Man erinnert sich: „Copyleft garantiert, dass jeder Nutzer Freiheit hat."[1], heißt es auf Seiten der FSF. Es kann und soll hier nicht abgestritten werden, dass proprietäre Software ein Machtinstrument darstellt, welches seine Benutzer gewissermaßen kontrolliert. Dies zu dementieren war allerdings auch nicht die Absicht dieser Arbeit; vielmehr sollte aufgezeigt werden, dass die Form der Software, die ein Nutzer wählt, nicht ausschlaggebend dafür ist, wie frei er in seinem (Computer-)Handeln ist. Es hat sich nämlich gezeigt, dass diese von Stallman und Weiteren angestrebte Freiheit in der Synthese mit der Hardware bestimmte Einschränkungen erfährt. Sie tut dies in den grundlegenden Stufen, die die Software bis zu ihrer ‚Vollendung' nimmt: der Programmierung, der Ausführung und der Bedienung. Deshalb soll die Frage nach der prinzipiellen Möglichkeit von Freiheit abschließend aufgeworfen werden. Dass es sich hierbei um eine komplexe Fragestellung handelt, mach Krajewski deutlich:

> Wer ist der Herr im Haus der Kalküle? Die zentrale Recheneinheit der Maschine, das heißt die königliche CPU, der alle anderen auf's Wort folgen? Der Operateur an der Konsole der *mainframe*, der Lochkarten kodierende Programmierer, der Endanwender vor dem eigens nach ihm benannten *graphical user interface*? Oder eher dessen virtuelles Doppel, der *client*?[2]

[1] Free Software Foundation: „Copyleft".
[2] Krajewski: „Dienstleistungsagenturen", S. 147.

Es soll an dieser Stelle kein Abwägen der einzelnen Positionen gegeneinander erfolgen, mit dem Ziel, einer Instanz die universale Macht zuzuschreiben. Ruft man sich allerdings noch einmal das Ausgangszitat Stallmans in Erinnerung, welches für diese Arbeit namensgebend war und davon ausging, dass jeder User uneingeschränkt die Kontrolle über seine Computernutzung besitzen sollte, lässt sich ein Punkt sehr deutlich machen. Im Hinblick auf die Vorrangigkeit der Hardware, die in einigen Teilen dieser Arbeit offenbart wurde, kann diese Aussage nur abgelehnt werden. Denn Stallmans Haltung: „With free software, the users control the program, both individually and collectively. So they control what their computers do […].“[3] erscheint nun zweifelsohne hinfällig. Die logische Konsequenz – im Wortlaut Stallmans – muss vielmehr lauten: We simply cannot have control of our own computing. Zumindest können wir über keine uneingeschränkte Kontrolle verfügen und diese erst recht nicht durch die Benutzung einer Freien Software alleine erhalten, die ja für die Bewegung der Freien Software das einzige Kriterium darstellt.[4]

In Bezug auf Stallmans Orientierung war insbesondere das Vorbild der Hacker Ethik zu nennen – ein Begriff, welchen er selbst geprägt hat und womit er grundsätzlich den uneingeschränkten Zugang zu allen Möglichkeiten der Maschine sowie die Wertschätzung von Informationen als freies, kollektives Gut meint. Es war sein Anliegen, diese moralischen Einstellungen nach über 20 Jahren zurück in die derzeitige Computerpraxis zu holen. Deswegen ist es gerade der Hacker, der für ihn die zentrale Rolle in der Geschichte von Software (und Hardware) spielt, und in dessen Macht es im weiteren Sinne steht, eine Emanzipation des Nutzers zu bewirken. Es mag eine unanfechtbare Tatsache sein, dass der Hacker das technisch-mediale Apriori des späteren 20. Jahrhunderts deutlich verschoben hat. Offen bleibt hingegen, ob diese Verschiebung tatsächlich zur Emanzipation oder vielmehr zur Entthronung des Individuums geführt hat. Organisationen wie beispielsweise der *Homebrew Computer Club* haben stets Bemühungen darüber angestellt, den Computer in solch einer Weise zu verbreiten, bis dieser für jedermann zugänglich sei. In der Tat wurde durch einige Teilnehmer innerhalb

[3] Stallman: „Free Software Is Even More Important Now“.
Interessanterweise bezieht auch Stallman sich explizit auf den Besitz von Kontrolle, wenn er seinen Wunsch nach einer allgemeinen Nutzerfreiheit auszudrücken vermag, so lässt sich hier in Rückbezug auf den Diskurs der Dialektik feststellen.

[4] „Für die Freie-Software-Bewegung bedeutet freie Software eine ethisch unbedingt erforderliche, wesentliche Achtung vor der Freiheit der Nutzer.“ Stallman: „Warum Open Source das Ziel Freie Software verfehlt“.

dieses Verbundes an Wissen und Ideen eine Reihe technischer Innovationen geboren. Auch Steve Wozniak hat zu den zentralen Mitgliedern des Clubs gehört und so kann im Übrigen auch der Apple II – mit dem die massenhafte Verbreitung des Heimcomputers Realität geworden ist – im weiteren Sinne als solch ein Gerät bezeichnet werden, welches aus den Inspirationen der Organisation heraus entstanden ist. Was für Levy als einer der Höhepunkte der Hacker-Bestrebungen gilt, führte im Wesentlichen zur Hervorbringung eines Computers, der erstmals auch für ‚gewöhnliche', weniger technikaffine Nutzer geeignet und sogar vorgesehen war. Hieran lässt sich also gleichzeitig auch ein Niedergang der Hacker Ethik andeuten, denn: „[…] *Apple* führte vor, daß Computer von jedem Laien benutzbar sein können, ohne daß er sie verstehen muß."[5] Dies war ein wichtiger Schritt hin zu einem User, der von den Machenschaften der Maschine nichts wissen konnte, wollte oder sollte. Die historische Entwicklung des Hackertums ist demnach eher kritisch zu betrachten und weniger als absolut positiv hinzunehmen.

> So wie im 18. und noch im 19. Jahrhundert die Freiheit von Vielen gegen die Freiheit der Wenigen zu erkämpfen war, so scheinen im 20. und vor allem im 21. Jahrhundert neue Monopole der Kontrolle einen Kontrollverlust der Vielen zu erzeugen. Um uns aus dem Dilemma zu befreien, das eine Spirale von subjektiver Freiheit und objekthafter Kontrolle anrichtet, ist nicht nur ein neuer Begriff der Freiheit nötig, sondern auch eine Aufklärung über die Grammatik der Kontrolle.[6]

Auch das bereits erwähnte Autorengespann Metz und Seeßlen sieht die Entwicklung der Freiheit als problematisch an. Sie fordern konkret eine Aufklärung der Verhältnisse, die grundlegend darin bestehen könnte, die existierenden Machthierarchien als solche zur Kenntnis zu nehmen. Wie sich beispielsweise in Bezug auf den Protected Mode gezeigt hat, besteht die grundlegende Problematik darin, *schein*bare Zustände zu erzeugen. Dabei lässt sich gerade in diesem gesicherten Betriebsmodus eine Umkehrung aller Machthierarchien, gegen die ‚Levys' Hacker mit allen Mitteln gekämpft haben, feststellen. Sie haben sich schließlich stets für eine uneingeschränkte Zugänglichkeit der Hardware eingesetzt und autoritäre Haltungen dementsprechend befehdetet. Mit der Einführung des Protected Modes wird nun nicht nur zwischen zwei verschiedenen Betriebsmodi, sondern vielmehr auch zwischen verschiedenen Handlungsmöglichkeiten unterschieden, die – dies kommt noch erschwerend hinzu – nicht klar als solche kenntlich gemacht werden und zwischen denen nur schwer gewechselt werden kann. Dabei hat der

---

[5] Pias, Claus: „Der Hacker", in: Horn, Eva; Kaufmann, Stefan; Bröckling, Ulrich (Hrsg.): *Grenzverletzer. Von Schmugglern, Spionen und anderen subversiven Gestalten*. Berlin: Kadmos 2002, S. 248-270, hier S. 252.
[6] Metz; Seeßlen: „Über die Freiheit".

Protected Mode zur Folge, dass prinzipiell vorhandene technische Möglichkeiten gedrosselt werden. Auf der Basis der Hardware kehrt das altbekannte Machtsystem hier sozusagen zurück.

Wie sich gezeigt hat, ist der User in seiner Nutzung auf vielen Ebenen abhängig von den maschinellen Gegebenheiten, die sich ihm darbieten. Schlussendlich ist es demnach ohne Belang, ob auf seinem Computer die Verwendung einer Open- oder Closed-Source-Software stattfindet. Denn in beiden Fällen bleibt die vollkommene Freiheit in der Nutzung eine Utopie. Trotz alldem, so soll hier verdeutlicht werden, ist das Konzept der Freien Software, wie auch der Open-Source-Software, durchaus sinnvoll, indem es dem Computeruser ein gewisses Maß an Respekt entgegenbringt. Es muss allerdings beachtet werden, dass sich die Freiheit, die ihm hierbei gewährt wird, nur innerhalb eines fest abgegrenzten Handlungsspielraums entfalten kann. Schließlich ist die Frage nach der Freiheit von Software immer auch eine Frage des Diskurses. Dies betonen Metz und Seeßlen: „Wer die Diskurse beherrscht, beherrscht auch die Menschen. Macht bedeutet nicht nur, die Freiheit anderer Menschen einschränken zu können, sondern auch zu bestimmen, was das eigentlich ist: Freiheit. Und welche Art von Freiheit gut ist und welche schlecht.“[7] Denn so kann das, was die einen als Fremdbestimmung wahrnehmen, für die anderen in einer Befreiung bestehen.

8.2 Versuch einer Neudefinition: Freie Software 2.0

„Freiheit … Kontrolle … Was für ein Dilemma.“[8]

Diese Arbeit soll es sich nicht bloß zur Aufgabe gemacht haben, die Existenz einer Freien Software zu dekonstruieren bzw. Stallmans Auffassung derselben zu kritisieren, sondern darüber hinaus auch Überlegungen darüber anzustellen, inwiefern es eine Alternative zur Freien Software bzw. für die ‚Befreiung‘ des Computernutzers geben könnte. Es soll also die Möglichkeit einer *Freien Software 2.0* verhandelt werden: Wie müsste Software gestaltet sein, damit sie im ‚richtigen‘ Kontext als frei bezeichnet werden kann? Und vielmehr: Kann es diese Form der Software überhaupt geben?

[7] Metz; Seeßlen: „Über die Freiheit“.
[8] Metz; Seeßlen: *Freiheit und Kontrolle*, S. 12.

Hierbei muss sich zunächst die Frage danach gestellt werden, was Medienkompetenzen in gegenwärtigen Zeiten bedeuten. Es ist offensichtlich, dass sich diese im Hinblick auf die Nutzung von Software mit fortschreitender Zeit verändert haben und sich stets weiterentwickeln werden. Stellt man einen Vergleich zwischen der Nutzung der frühen Hacker in den 60er Jahren und der heutigen Nutzung an, so wird in Bezug auf die massenhafte Verbreitung des Computers deutlich, dass der höhere Bedarf an Software auch ein entsprechend hohes Maß an Produzenten erfordert. Vor allem aber fällt auf, dass die Notwendigkeit, sich die Funktionsweisen des Rechners anzueignen, in der heutigen Zeit völlig weggefallen ist. Der moderne User kann auf – freie oder nicht-freie – Software zugreifen, ohne sich jemals mit der Funktionsweise der Hardwareebene auseinandersetzen zu müssen oder mit dieser nur annähernd in Berührung zu kommen. Das Ergebnis aus dieser Entwicklung ist eine ausgeprägte Differenzierung zwischen einem Experten, der sich die komputatorische Situation zu eigen machen und diese verändern kann, und einem Amateur, der das ihm Gegebene lediglich benutzen kann. Diese Unterscheidung gab es zu früheren Zeiten noch nicht; wer den Computer nutzen wollte, musste sich mit dessen Architektur vertraut machen und Programme nach eigenem Ermessen selbst schreiben.

Als aktuelles Beispiel soll die Thematik von Programmierunterricht in der Schule betrachtet werden. Schließlich gilt das Programmieren lernen „als Symbol dafür, mit den Herausforderungen der digitalen Welt besser zurechtzukommen.“[9] Im Rahmen dieser Arbeit ist es allerdings nur konsequent, zu hinterfragen, ob man tatsächlich als kompetent gelten kann, wenn man mit der Software von Computern, nicht aber mit deren Hardware umgehen kann. Oder mit den Worten Frank Hartmanns:

> Beherrscht der Mensch nun die Technik, wenn er beispielsweise als Automechaniker in die letzten Geheimnisse von Motor, Kupplung, Bremssystem und Servolenkung vordringt? Oder aber, wenn er als Fahrer einfach vom Gas geht und zurückschaltet, um rechtzeitig die Kurve zu kriegen?[10]

Allgemein anerkannt ist wohl eher letzterer vorgebrachter Vorschlag. Dass dieser definitiv zu bemängeln ist, macht unter anderem Sascha Lobo deutlich: „Wer programmieren kann, kann programmieren – was aber dringend und immer schmerzlicher

[9] Lobo, Sascha: „Digitalverständnis von Schülern. Programmieren lernen hilft nicht“, in: *Spiegel Online*, 29.03.2017. http://www.spiegel.de/netzwelt/web/programmieren-in-der-schule-sollen-kinder-programmieren-lernen-kolumne-a-1140928.html (Abrufdatum: 07.03.2018).
[10] Hartmann: „Vom Sündenfall der Software“.

fehlt, ist ein Verständnis der Zusammenhänge einer digital vernetzten Welt und nicht ihrer kleinsten Bausteine."[11] Um dieses tiefes Verständnis der Zusammenhänge innerhalb eines Systems zu erlangen, kann es nicht ausreichend sein, die Bedienung eines Gerätes auf einer oberflächlichen Ebene – man rufe sich die Benutzeroberfläche in Erinnerung – meistern zu können. Vielmehr bedürfte es fundierten Hardware-Kenntnissen, die eine Nutzung des Computers auch ohne die Distanzierung der gängigen Software ermöglichen würden. Um dies zu erreichen, wäre die uneingeschränkte Kenntnis der Maschinensprache des jeweiligen Computers nötig. „Kittlers Ideal wäre wohl so etwas wie ein ‚Maschinenflüsterer', jemand, der den unbewußten maschinellen Code so beherrscht, daß er sie sehr wohl zur souveränen Anwendung und damit zur Beherrschbarkeit bringen kann."[12], fasst Hartmann die Kittlersche Perspektive zusammen.

Noch konsequenter wäre es, die Programmierung mittels direktem Zugriff auf die Hardwareebene zu praktizieren. Um eine spezifische Programmierung zu erreichen, müssten die Programme der Maschine dafür händisch eingeführt, also eine elektronische Neuverschaltung durchgeführt werden. ‚Software' müsste also im wahrsten Sinne physisch gedacht werden. Schließlich geht das grundlegende Problem darauf zurück, dass zwischen User und Computer ein Gefüge aus Software steht, das in vielerlei Hinsicht vermittelt und verschleiert. Es wäre demnach anzuzweifeln, inwiefern eine Trennung von Hard- und Software überhaupt produktiv ist – und demnach wäre auch die Möglichkeit einer *Freien Software 2.0* hinfällig, denn um die vollkommene Freiheit zu erreichen, müsste die Computernutzung ganz ohne Software im heutigen Sinne erfolgen. Die seit den 50er Jahren etablierte Trennung der beiden Komponenten ist schließlich künstlich und mit der Absicht einer Arbeitsteilung erzeugt worden, wie in der kurzen Darlegung des *Mark I* deutlich wurde. Stellt man dem beispielweise Turings Entwurf einer Rechenmaschine aus den 30er Jahren gegenüber, so lässt sich hier noch keine wirkliche Trennung der Software von der Hardware feststellen.[13] Medienökonomisch, im Hinblick auf die Effizienz, gesehen ist es aufgrund der Komplexität moderner Computer wohl als äußert fragwürdig einzuschätzen, die physische Neuverschaltung als gängige *Programmier*praxis wiedereinzuführen. Ein mögliches, in der Praxis angewendetes

[11] Lobo: „Digitalverständnis von Schülern".
[12] Hartmann: „Vom Sündenfall der Software".
[13] Vgl.: Turing, Alan: „On Computable Numbers, with an Application to the Entscheidungsproblem", in: *Proceedings of the London Mathematical Society*, Band 42, Heft o. A. (1936), S. 230-265.

Beispiel stellt dagegen der sogenannte *Field Programmable Gate Array* (FPGA) dar. Dabei handelt es sich um einen mikroelektronischen Baustein, der – im Gegensatz zu einem regulär verwendeten Logikgatter – eine undefinierte Funktion aufweist und per se über keinen Speicher verfügt, sodass er wiederholt frei programmierbar ist. In seiner unerweiterten Form kann ein FPGA keine Software ausführen; vielmehr ermöglicht er die symbolische Programmierung der Hardware auf mikroelektronischer Ebene, indem die Schaltungsstruktur mittels Hardware-Beschreibungssprache formuliert wird.[14] Letztlich bleibt es jedoch zweifelhaft, ob dieses und mögliche weitere Beispiele überhaupt die Möglichkeit darstellen, sich als Nutzer dem ‚programmierten' und ‚programmierenden' System entziehen zu können. Denn wie bereits Winthrop-Young festgestellt hat: „Der Mensch ist das Wesen, das immer wieder dazu programmiert wird, seine Programmierungen zu verkennen."[15]

[14] Vgl.: Dondo Gazzano, Julio Daniel [u.a]: *Field-Programmable Gate Array (FPGA) Technologies for High Performance Instrumentation*. Hershey: Engineering Science Reference 2016, S. xvff.
„Andererseits sind sie komplex, langsam und teuer, was sie außer für Nischenanwendungen unattraktiv macht.", so Tanenbaum. Tanenbaum: *Computerarchitektur*, S. 605.
[15] Winthrop-Young: *Friedrich Kittler zur Einführung*, S. 170.

9 Ausblick

Nachdem die Frage nach der Freiheit von Software im Rahmen des Möglichen geklärt wurde, soll ihre Relevanz abschließend in einen aktuellen Kontext gestellt werden. Vor wenigen Jahren wurde ein Diskurs von Claus Pias angestoßen, der sich deutlich dafür aussprach, das Kittlersche Œuvre müsse – entgegen der Mehrheit seiner heutigen Anwendung – stets technikhistorisch lokalisiert werden.[1] Während Kittlers Medientheorie der 80er Jahre nämlich zwangsläufig von der Erfahrung mit Home- und Personalcomputern geprägt war, sieht sich der moderne User gänzlich anderen Techniken konfrontiert:

> Und für unsere Gegenwart liegt die Sache noch einmal grundlegend anders, weil sich Einzelgeräte mittlerweile weitgehend in ‚Dienste' aufgelöst haben – und mit ihnen die originären Zweckbestimmungen, Ästhetiken oder Subjektivierungsformen verschwunden sind, die man Einzelmedien in den Achtzigern noch zuschreiben wollte.[2]

So lässt sich zunächst festhalten, dass an die Stelle eines eigenständigen Computers das Smartphone getreten ist – das schon mit seinem Namen eine Verwirrung bezweckt, indem es vorgibt, mehr Telefon als Computer zu sein. Auch Pias betont, dass dies nicht der Wahrheit entspricht:

> Und was den ‚Mißbrauch' betrifft, ließe sich konstatieren, dass ein Smartphone eine Black Box ist, die ein Telefon, ein Fotoapparat, ein Walkman, eine Taschenlampe, ein Wecker, ein Weltempfänger, ein Taschenrechner, eine Textverarbeitung, eine Tabellenkalkulation, ein Videospiel, ein Atlas, ein Lexikon und alles mögliche andere *ist*.[3]

Dass, wie Pias es ausdrückt, prinzipiell alles andere mit dem Benutzen eines Smartphones möglich ist, macht es äußert schwierig, Grenzen zu erkennen. An die Stelle eines Einzel- ist demnach ein Vielfachgerät getreten.

Mit dem Einzug des Internets in jede erdenkliche Lücke des alltäglichen Lebens wird die Komplexität noch weiter gesteigert. Die Vorherrschaft des Smartphones in Verbindung mit der digitalen Vernetzung macht es möglich, dass ein User keinen Einfluss auf die Kommunikation seines Gerätes zu weiteren Geräten, Servern etc. nehmen kann. Dabei

---

[1] Vgl.: Pias, Claus: „Friedrich Kittler und der ‚Mißbrauch von Heeresgerät'. Zur Situation eines Denkbildes 1964 – 1984 – 2014", in: *Merkur. Deutsche Zeitschrift für europäisches Denken*. Band 69, Heft 4 (2015), S. 31-44, hier S. 31f.
[2] Ebd., S. 33.
[3] Ebd., S. 34. Pias bezieht sich in seiner Ausführung auf Kittlers Rede vom ‚Mißbrauch von Heeresgerät', mit welcher dieser die Verbindung von Funkgerät und Grammofon und damit die Umfunktionierung derselben zu einem Prototyp des Rundfunks ergibt. Vgl.: Ebd., S. 36.

erschien auch das Internet einst als das Medium, welches eine freie Verbreitung von Informationen, zugänglich für jeden seiner Nutzer, darstellte. Doch auch hier muss die Existenz einer Kehrseite anerkannt werden:

> Bevor Google, Amazon, NSA oder Microsoft ihre Informationsmonopole so schamlos ausstellten, vermittelte das Internet die Ahnung einer völlig neuen, völlig flüssigen Form der Demokratie. Einer Demokratie unter dem Motto: Jeder kontrolliert jeden. In aller Freiheit.[4]

Es stellt sich hier erneut die Frage danach, welchen Nutzen Freie Software bringt, wenn man es statt mit dem Einzelmedium Computer vielmehr mit vernetzten digitalen Geräten oder Webdiensten zu tun hat. Bereits eine oberflächliche Betrachtung lässt die Vermutung aufkommen, als werde der Nutzer erneut – und vielleicht sogar verstärkt – zum bloßen *Be*nutzer degradiert. Denn es erscheint immer schwerer, als Konsument den Überblick über die von der Industrie gebotenen Angebote oder gar die umfassende Kontrolle über die eigenen Aktionen zu behalten.

Als weiterer, potenziell wichtiger Aspekt soll an dieser Stelle die Künstliche Intelligenz genannt werden. Vor etwa einem Jahr hat sich Google-Mitgründer Sergey Brin zu ihrer Unvorhersehbarkeit geäußert: „What can these things do? We don't really know the limits. It has incredible possibilities. I think it's impossible to forecast accurately."[5] Dass die vollständige Automatisierung von intelligentem Maschinenverhalten weitreichende Folgen mit sich bringen wird, muss nicht weiter erläutert werden. Dass sich jedoch gerade der Mitentwickler einer der weltweit größten Internetdienstleister dazu äußert, ihre Grenzen und Möglichkeiten nicht einschätzen zu können, gibt Anlass zur Reflexion. Wie sich ihre weitere Entwicklung gestalten wird, bleibt in den folgenden Jahren zu beobachten. Ebenso könnte es von Interesse sein, zu verfolgen, wie sich aus den gängigen Webdiensten heraus das sogenannte *Cloud Computing* weiterentwickelt und künftig Infrastrukturen ausgebaut werden, die keine lokalen Speicherdienste mehr beanspruchen.[6] Die Andeutung all dessen lässt vermuten, dass die Forderung nach Freier Software wohl weiter an Relevanz verlieren wird. Um die Kontrolle über die Operationen der Geräte zu bewahren, müssten dann andere Fragestellungen diskutiert werden, die mit Softwarelizenzen nur noch am Rande zu tun haben können.

---

[4] Metz; Seeßlen: „Über die Freiheit".
[5] Zitiert nach: Chainey, Ross: „Google co-founder Sergey Brin: I didn't see AI coming", in: *The World Economic Forum*, 19.01.2017. https://www.weforum.org/agenda/2017/01/google-sergey-brin-i-didn-t-see-ai-coming/ (Abrufdatum: 12.03.2018).
[6] Vgl.: Fischer; Hofer: *Lexikon der Informatik*, S. 993.

Abbildungsverzeichnis

Literaturverzeichnis

Aho, Alfred V. [u.a.]: *Compilers. Principles, Techniques, & Tools*. Boston [u.a.]: Pearson Addison-Wesley 2007.

Asendorpf, Dirk: „Error“, in: *ZEIT ONLINE*, 26.07.2017. http://www.zeit.de/2017/31/computer-software-fehler-systeme (Abrufdatum: 13.12.2017).

Backus, John: „Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs“, in: *Communications of the ACM*, Band 21, Heft 8 (1978), S. 613-641.

Bauer, Friedrich L.: *Kurze Geschichte der Informatik*. München [u.a]: Fink 2009.

Baustädter, Birgit: „TU Graz-Forscher entdecken schwerwiegende IT-Sicherheitslücke“, in: *TU Graz news*, 04.01.2018. https://www.tugraz.at/tu-graz/services/news-stories/medienservice/einzelansicht/article/schwere-sicherheitsluecke-tu-graz-forscher-zentral-beteiligt/ (Abrufdatum: 16.02.2018).

Buxmann, Peter; Diefenbach, Heiner; Hess, Thomas: *Die Softwareindustrie. Ökonomische Prinzipien, Strategien, Perspektiven*. Berlin/Heidelberg: Springer 2011.

Campbell-Kelly, Martin; Aspray, William: *Computer. A history of the information machine*. New York: Basic Books 1996.

Ceruzzi, Paul E.: *A history of modern computing*. Cambridge/London: MIT Press 2003.

Chainey, Ross: „Google co-founder Sergey Brin: I didn’t see AI coming“, in: *The World Economic Forum*, 19.01.2017. https://www.weforum.org/agenda/2017/01/google-sergey-brin-i-didn-t-see-ai-coming/ (Abrufdatum: 12.03.2018).

Chomsky, Noam: „Three Models for the Description of Language“, in: *IRE Transactions on Information Theory*, Band 2, Heft 3 (1956), S. 113-124.

Chun, Wendy H. K.: „On Software, or the Persistence of Visual Knowledge“, in: *Grey Room*, Band 18, Heft o. A. (2005), S. 26-51.

Coy, Wolfgang: *Aufbau und Arbeitsweise von Rechenanlagen. Eine Einführung in Rechnerarchitektur und Rechnerorganisation für das Grundstudium der Informatik*. Braunschweig/Wiesbaden: Vieweg 1992.

Cramer, Florian; Fuller, Matthew: „Interface“, in: Fuller, Matthew (Hrsg.): *Software Studies. A Lexicon*. Cambridge/London: MIT Press 2008, S. 149-152.

Dijkstra, Edsger W.: „The Humble Programmer“, in: *Communications of the ACM*, Band 15, Heft 10 (1972), S. 859-866.

Dondo Gazzano, Julio Daniel [u.a.]: *Field-Programmable Gate Array (FPGA) Technologies for High Performance Instrumentation*. Hershey: Engineering Science Reference 2016.

Faggin, Federico: „How we made the microprocessor“, in: *Nature Electronics*, 08.01.2018. https://www.nature.com/articles/s41928-017-0014-8 (Abrufdatum: 29. 01.2018).

Fischer, Peter; Hofer, Peter: *Lexikon der Informatik*. Berlin/Heidelberg: Springer 2011.

Frabetti, Federica: *Software Theory. A Cultural and Philosophical Study*. London/New York: Rowman & Littlefield International 2015.

Free Software Foundation, Website des GNU-Systems. https://www.gnu.org/ (Abrufdatum: 03.11.2017).

– „Copyleft. Was ist das?“, in: https://www.gnu.org/licenses/copyleft.de.html (Abrufdatum: 01.11.2017).
– „Freie Software. Was ist das?“, in: https://www.gnu.org/philosophy/free-sw (Abrufdatum: 23.10.2017).
– „Geschichte des GNU-Systems“, in: https://www.gnu.org/gnu/gnu-history (Abrufdatum: 05.01.2018).
– „GNU/Linux: Häufig gestellte Fragen. Warum wird das benutzte System GNU/Linux genannt, nicht ‚Linux‘?“, in: https://www.gnu.org/gnu/gnu-linux-faq.de.html#why (Abrufdatum: 05.01.2018).
– „Initial Announcement“, in: https://www.gnu.org/gnu/initial-announcement.en.html (Abrufdatum: 05.01.2018).
– „Kategorien freier und unfreier Software“, in: https://www.gnu.org/philosophy/categories.html (Abrufdatum: 23.10.2017).
– „Lizenzen“, in: https://www.gnu.org/licenses/ (Abrufdatum: 01.11.2017).

Fuller, Matthew (Hrsg.): *Software Studies. A Lexicon*. Cambridge/London: MIT Press 2008.

Grassmuck, Volker: „Das Ende der Universalmaschine“, in: Pias, Claus (Hrsg.): *Zukünfte des Computers*. Zürich/Berlin: diaphanes 2004, S. 241-269.

– *Freie Software. Zwischen Privat- und Gemeineigentum*. Bonn: Bundeszentrale für

politische Bildung 2004.

Hagen, Wolfgang: „Der Stil der Sourcen. Anmerkungen zur Theorie und Geschichte der Programmiersprachen“, in: Warnke, Martin; Coy, Wolfgang; Tholen, Georg C. (Hrsg.): *HyperKult. Geschichte, Theorie und Kontext digitaler Medien*. Basel/Frankfurt am Main: Stroemfeld 1997, S. 33-68.

Hartmann, Frank: „Friedrich Kittler“, in: Sander, Uwe; von Gross, Friederike; Hugger, Kai-Uwe (Hrsg.): *Handbuch Medienpädagogik*. Wiesbaden: VS Verlag für Sozialwissenschaften 2008, S. 251-256.

– „Vom Sündenfall der Software“, in: *Telepolis*, 22.12.1998. https://www.heise.de/tp/features/Vom-Suendenfall-der-Software-3601396.html (Abrufdatum: 04.10.2017).

Hedtstück, Ulrich: *Einführung in die Theoretische Informatik. Formale Sprachen und Automatentheorie*. Berlin/Boston: De Gruyter 2012.

Hegel, Georg Wilhelm Friedrich: *Enzyklopädie der philosophischen Wissenschaften I. Werke 8*. Erschienen in: *Werke in 20 Bänden mit Registerband*, editiert von Moldenhauer, Eva; Michel, Karl Markus. Frankfurt am Main: Suhrkamp 1989.

– *Wissenschaft der Logik II. Werke 6*. Erschienen in: *Werke in 20 Bänden mit Registerband*, editiert von Moldenhauer, Eva; Michel, Karl Markus. Frankfurt am Main: Suhrkamp 1990.

Himanen, Pekka: *Die Hacker-Ethik und der Geist des Informations-Zeitalters*. München: Riemann 2001.

Höltgen, Stefan: „‚All Watched Over by Machines of Loving Grace‘. Öffentliche Erinnerungen, demokratische Informationen und restriktive Technologien am Beispiel der ‚Community Memory‘“, in: Reichert, Ramón (Hrsg.): *Big Data. Analysen zum digitalen Wandel von Wissen, Macht und Ökonomie*. Bielefeld: transcript 2014, S. 385-403.

Honneth, Axel (Hrsg.): *Dialektik der Freiheit. Frankfurter Adorno-Konferenz 2003*. Frankfurt am Main: Suhrkamp 2005.

Hülsbömer, Simon: „Der x86-Prozessor wird 30 – wie Intel dank IBM alle Gipfel stürmte“, in: *COMPUTERWOCHE Online*, 23.06.2008. https://www.computerwoche.de/a/der-x86-prozessor-wird-30-wie-intel-dank-ibm-alle-gipfel-stuermte,1866928 (Abrufdatum: 14.02.2018).

IBM Corporation: „Chronological History of IBM. 1960s“, in: IBM Archives Online. http://www-03.ibm.com/ibm/history/history/decade_1960.html (Abrufdatum: 20.11. 2017).

Intel Corporation: „80286 and 80287 Programmer's Reference Manual“, 1987, in: http://bitsavers.trailing-edge.com/components/intel/80286/210498-005_80286_and_ 80287_Programmers_Reference_Manual_1987.pdf (Abrufdatum: 16.03.2018).

– „Intel® 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture“, in: Intel Software Developer Zone, 2017. https://software.intel.com/ sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf (Abrufdatum: 13.12.2017).
– „Intel Timeline. A History of Innovation“, in: Intel Website. https://www.intel.com/content/www/us/en/history/historic-timeline.html (Abrufdatum: 08.12.2017).

Janschitz, Mario: „UI vs UX: ‚So sieht es aus' vs. ‚So fühlt es sich an'“, in: *t3n Online*, 03.09.2015. https://t3n.de/news/ui-ux-sieht-vs-fuehlt-635734/ (Abrufdatum: 26.02. 2018).

Kittler, Friedrich A.: „Buchstaben → Zahlen → Codes“, in: Brüning, Jochen; Knobloch, Eberhard: *Die mathematischen Wurzeln der Kultur. Mathematische Innovationen und ihre kulturellen Folgen*. München: Wilhelm Fink 2005, S. 65-76.

– „Computeranalphabetismus“. In: Matejovski, Dirk; Kittler, Friedrich (Hrsg.): *Literatur im Informationszeitalter*. Frankfurt/New York: Campus Verlag 1996, S. 237-251.
– *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993.
– „Es gibt keine Software“, in: Ders.: *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993, S. 225-242.
– *Grammophon, Film, Typewriter*. Berlin: Brinkmann & Bose 1986.
– „Hardware, das unbekannte Wesen“, in: Krämer, Sybille (Hrsg.): *Medien, Computer, Realität. Wirklichkeitsvorstellungen und Neue Medien*. Frankfurt am Main: Suhrkamp 1998, S. 119-132.
– „Protected Mode“, in: Ders.: *Draculas Vermächtnis. Technische Schriften*. Leipzig: Reclam 1993, S. 208-224.

Korb, Joachim: „Geschichte der Softwareprogrammierung: ‚Freie Software für Freiheit und Gerechtigkeit'“, in: *Perspektive'89*, 21.12.2006. http://perspektive89.com/

2006/12/21/geschichte_der_softwareprogrammierung_freie_software_fur_freiheit_und_gerechtigkeit (Abrufdatum: 05.11.2017).

Krajewski, Markus: „Dienstleistungsagenturen. Zur Delegation von Handlungsmacht zwischen Subalternen und Software-Services“, in: Mersch, Dieter; Paech, Joachim (Hrsg.): *Programm(e)*. Zürich/Berlin: diaphanes 2004, S. 125-157.

Kroll, Joachim: „Sicherheitslücke in Prozessoren. Was hinter ‚Meltdown‘ und ‚Spectre‘ steckt“, in: *Computer&AUTOMATION Online*, 08.01.2018. http://www.computer-automation.de/steuerungsebene/safety-security/artikel/149237/ (Abrufdatum: 14.02.2018).

Levy, Steven: *Hackers. Heroes of the computer revolution*. New York: Anchor Press/Doubleday 1984.

Lipp, Moritz [u.a.]: „Meltdown“, in: https://meltdownattack.com/meltdown.pdf (Abrufdatum: 14.02.2018).

Lobo, Sascha: „Digitalverständnis von Schülern. Programmieren lernen hilft nicht“, in: *Spiegel Online*, 29.03.2017. http://www.spiegel.de/netzwelt/web/programmieren-in-der-schule-sollen-kinder-programmieren-lernen-kolumne-a-1140928.html (Abrufdatum: 07.03.2018).

Mandl, Peter: *Grundkurs Betriebssysteme. Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung*. Wiesbaden: Springer Vieweg 2014.

Metz, Markus; Seeßlen, Georg: *Freiheit und Kontrolle. Die Geschichte des nicht zu Ende befreiten Sklaven*. Berlin: Suhrkamp 2017.

– „Über die Freiheit. Die dialektische Beziehung von Freiheit und Kontrolle“, in: *Deutschlandfunk Online*, 05.10.2014. http://www.deutschlandfunk.de/ueber-die-freiheit-die-dialektische-beziehung-von-freiheit.1184.de.html?dram:article_id=294982 (Abrufdatum: 12.01.2018).

Möller, Erik: *Die heimliche Medienrevolution. Wie Weblogs, Wikis und freie Software die Welt verändern*. Hannover: Heise 2006.

Naumann, Friedrich: *Vom Abaskus zum Internet. Die Geschichte der Informatik*. Darmstadt: Primus 2001.

Open Source Initiative: „History of the OSI“, in: https://opensource.org/history

(Abrufdatum: 05.01.2018).

Open Source Initiative: „The Open Source Definition“, in: https://opensource.org/osd (Abrufdatum: 05.01.2018).

Osterloh, Margit; Rota, Sandra: „Open source software development. Just another case of collective invention?“, in: *Research Policy*, Band 36 (2007), S. 157-171.

Perkel, Jeffrey: „Democratic Databases: Science on GitHub“, in: *Nature*, Band 538, Heft 7623 (2016), S. 127-129.

Pfaffenberger, Bryan: „The Rhetoric of Dread. Fear, Uncertainty, and Doubt (FUD) in Information Technology Marketing“, in: *Knowledge, Technology & Policy*, Band 13, Heft 3 (2000), S. 78-92.

Pfeiffer Consulting: „User Interface Friction Research. Research Report“, in: http://www.pfeifferreport.com/v2/wp-content/uploads/2013/05/UIF_Rep.pdf (Abruf-datum: 26.02.2018).

Pias, Claus: „‚Children of the revolution‘. Video-Spiel-Computer als Kreuzungen der Informationsgesellschaft“, in: Ders. (Hrsg.): *Zukünfte des Computers*. Zürich/Berlin: diaphanes 2004, S. 217-240.

– „Der Hacker“, in: Horn, Eva; Kaufmann, Stefan; Bröckling, Ulrich (Hrsg.): *Grenzverletzer. Von Schmugglern, Spionen und anderen subversiven Gestalten*. Berlin: Kadmos 2002, S. 248-270.

– „Die Pflichten des Spielers. Der User als Gestalt der Anschlüsse“, in: Warnke, Martin; Coy, Wolfgang; Tholen, Georg C. (Hrsg.): *HyperKult II. Zur Ortsbestimmung analoger und digitaler Medien*. Bielefeld: transcript 2005, S. 313-341.

– „Friedrich Kittler und der ‚Mißbrauch von Heeresgerät‘. Zur Situation eines Denkbildes 1964 – 1984 – 2014“, in: *Merkur. Deutsche Zeitschrift für europäisches Denken*, Band 69, Heft 4 (2015), S. 31-44.

Pircher, Wolfgang: „Das Wissen des Kapitals und der Software-Anarchismus. Ein Kommentar zum GNU-Manifest“, in: Pias, Claus (Hrsg.): *Zukünfte des Computers*. Zürich/Berlin: diaphanes 2004, S. 207-215.

Rau, Thomas: „Test: Das bringt ein schneller Prozessor“, in: *PC-WELT Online*, 04.10.2012. https://www.pc welt.de/produkte/Dual-und-Quad-Core-Test-Das-bringt-ein-schneller-Prozessor-6699326.html (Abrufdatum: 16.03.2018).

Raymond, Eric S.: *The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol: O'Reilly 1999.

Rechenberg, Peter: *Was ist Informatik? Eine allgemeinverständliche Einführung*. München: Hanser 1994.

Ritchie, Dennis M.; Thompson, Ken: „The UNIX Time-Sharing System", in: *The Bell System Technical Journal*, Band 57, Heft 6 (1978), S. 1905-1929.

Rundnagel, Regine: „Software-Ergonomie und Benutzungsfreundlichkeit", in: *ergo-online*, http://www.ergo-online.de/site.aspx?url=html/software/grundlagen_der_software_ergon/software_ergonomie.htm (Abrufdatum: 03.03.2018).

Schief, Rudolf: *Einführung in die Mikroprozessoren und Mikrocomputer. Am Beispiel der Mikroprozessoren 8080, 8085, Z80, 8086/8088, 80286, 80386*. Tübingen: Attempto 1991.

Schweppenhäuser, Gerhard: *Ethik nach Auschwitz. Adornos negative Moralphilosophie*. Hamburg: Argument-Verlag 1993.

Sommerville, Ian: *Software Engineering*. München: Pearson 2012.

Spiegel, André: „Kopien verändern die Welt: Die Befreiung der Information", in: *t3n Online*, 05.09.2007. http://t3n.de/magazin/befreiung-information-kopien-verandern-welt-220134/ (Abrufdatum: 22.11.2017).

Stallman, Richard: „Free Software Is Even More Important Now", in: Website des GNU-Systems. https://www.gnu.org/philosophy/free-software-even-more-important.en.html (Abrufdatum: 20.12.2017).

– „Warum Open Source das Ziel Freie Software verfehlt", in: Website des GNU-Systems. https://www.gnu.org/philosophy/open-source-misses-the-point.de.html (Abrufdatum: 23.10.2017).

Tanenbaum, Andrew S.: *Computerarchitektur. Strukturen – Konzepte – Grundlagen*. München: Pearson 2006.

Technische Universität Graz: „Meltdown and Spectre. Vulnerabilities in modern computers leak passwords and sensitive data", in: https://spectreattack.com/ (Abrufdatum: 14.02.2018).

Thies, Klaus-Dieter: *Das 80186 Handbuch*. Düsseldorf: SYBEX 1986.

Thung, Ferdian [u.a.]: „Network Structure of Social Coding in GitHub“, in: *17th European Conference on Software Maintenance and Reengineering (CSMR 2013), März 2013, Genua, Italien*, S. 1-4.

Tukey, John W.: „The Teaching of Concrete Mathematics“, in: *The American Mathematical Monthly*, Band 65, Heft 1 (1958), S. 1-9.

Turing, Alan: „On Computable Numbers, with an Application to the Entscheidungsproblem“, in: *Proceedings of the London Mathematical Society*, Band 42, Heft o. A. (1936), S. 230-265.

Turner, Fred: *From Counterculture to Cyberculture. Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*. Chicago, London: The University of Chicago Press 2006.

Vogt, Carsten: *Nebenläufige Programmierung. Ein Arbeitsbuch mit UNIX/Linux und Java*. München: Hanser 2012.

de Vos, Lu: „Dialektik“, in: Cobben, Paul [u.a.] (Hrsg.): *Hegel-Lexikon*. Darmstadt: Wissenschaftliche Buchgesellschaft 2006, S. 142-143.

Wichmann, Thorsten: *Linux- und Open-Source-Strategien*. Berlin, Heidelberg: Springer 2005.

Wiebe, Frank: „Fichte und die Dialektik der Freiheit“, in: *SciLogs*, 13.09.2012. https://scilogs.spektrum.de/gute-geschaefte/fichte-und-die-dialektik-der-freiheit/ (Abrufdatum: 12.01.2018).

Windeck, Christof: „Meltdown und Spectre im Überblick: Grundlagen, Auswirkungen und Praxistipps“, in: *heise online*, 19.01.2018. https://www.heise.de/security/meldung/Meltdown-und-Spectre-im-Ueberblick-Grundlagen-Auswirkungen-und-Praxistipps-3944915.html (Abrufdatum: 18.02.2018).

Winkler, Hartmut: *Docuverse. Zur Medientheorie der Computer*. München: Boer 1997.

Winthrop-Young, Geoffrey: *Friedrich Kittler zur Einführung*. Hamburg: Junius 2005.